USER MANUAL

KEITHLEY METRABYTE/ASYST/DAC

PDMA-32

* * * * *

The

PDMA-32 Parallel Digital Interface Board

with

Direct Memory Access

Revision A, - July 1988
Copyright © Keithley MetraByte Corp. 1988
Part Number: 24847

KEITHLEY METRABYTE CORPORATION

440 MYLES STANDISH BLVD., Taunton, MA 02780
TEL. 508/880-3000, FAX 508/880-0179

Warranty Information

All products manufactured by Keithley MetraByte are warranted against defective materials and worksmanship for a period of one year from the date of delivery to the original purchaser. Any product that is found to be defective within the warranty period will, at the option of Keithley MetraByte, be repaired or replaced. This warranty does not apply to products damaged by improper use.

Warning

Keithley MetraByte assumes no liability for damages consequent to the use of this product. This product is not designed with components of a level of reliability suitable for use in life support or critical applications.

Disclaimer

Information furnished by Keithley MetraByte is believed to be accurate and reliable. However, the Keithley MetraByte Corporation assumes no responsibility for the use of such information nor for any infringements of patents or other rights of third parties that may result from its use. No license is granted by implication or otherwise under any patent rights of Keithley MetraByte Corporation.

Notes

Keithley MetraByte/Asyst/DAC is also referred to here-in as *Keithley MetraByte*.

Basic™ is a trademark of Dartmouth College.

IBM® is a registered trademark of International Business Machines Corporation.

PC, **XT**, **AT**, **PS/2**, and **Micro Channel Architecture**® (MCA) are trademarks of International Business Machines Corporation.

Microsoft® is a registered trademark of Microsoft Corporation.

Turbo C® is a registered trademark of Borland International.

WARRANTY

All products manufactured by MetraByte Corporation are warranted against defective workmanship and materials for a period of 1 year from the date of delivery to the original purchaser. Any product that is found to be defective within the warranty period will, at the option of MetraByte, be repaired or replaced. This warranty does not apply to products damaged by improper use.

! WARNING !

MetraByte Corporation assumes no liability to damages consequent to the use of this product. This product is not designed with components with a level of reliability suitable for use in life support systems.

MetraByte Corporation, 440 Myles Standish Boulevard, Taunton, Mass. 02780 U.S.A.
Phone: 508-880-3000 Telex: 503989 Fax: 508-880-0179 RBBS: 508-824-8659

Digitized by the Internet Archive
in 2025

https://archive.org/details/manualsplus_07913

Table of Contents

Section 1 INTRODUCTION 1-1
1.1 PDMA-32 Digital D.M.A. Interface Card Description 1-1
1.2 Accessories and Support 1-3

Section 2 INSTALLATION 2-1
2.1 Backing up the Disk 2-1
2.2 Hardware Installation 2-1

Section 3 UNDERSTANDING D.M.A. 3-1
3.1 What is D.M.A.? 3-1
3.2 The Mechanics of a D.M.A. Transfer 3-2
3.3 D.M.A. Structure of the PC/AT 3-3
3.4 The 8237 D.M.A. Controller 3-6
3.5 The D.M.A. Page Registers 3-11
3.6 Setting up a D.M.A. Transfer 3-12
3.7 D.M.A. Transfer Timing 3-14
3.8 Using Interrupts with D.M.A. 3-15
3.9 Determining Status of a D.M.A. Transfer 3-15

Section 4 REGISTER STRUCTURE & CONNECTIONS 4-1
4.1 I/O Map 4-1
4.2 Register Functions 4-2
4.2.1 Ports A & B 4-2
4.2.2 D.M.A. Control Register 4-3
4.2.3 Interrupt Control Register 4-4
4.2.4 8254 Timer 4-5
4.2.5 D.M.A. Level Select Register 4-6
4.2.6 Interrupt Level Register 4-7
4.2.7 Interrupt Status Register 4-7
4.3 Connector Pin Assignment 4-8
4.4 I/O Signal Functions 4-9

Section 5 PROGRAMMING 5-1
5.1 Programming PDMA-32 5-1
5.2 Loading the Machine Language CALL Routine PDMA32.BIN 5-1
5.3 Structure of the CALL Statement 5-3
5.4 Error Codes 5-5
5.5 CALL Modes 5-6
5.6 Mode 0: Initialization 5-7
5.7 Mode 1: Set up and Perform D.M.A. Transfer 5-8
5.8 Mode 2: Return Status 5-11
5.9 Mode 3: Set Timer 5-12
5.10 Mode 4: Digital Output 5-13
5.11 Mode 5: Digital Input 5-15

5.12 Mode 6: Auxiliary Output 5-17
5.13 Mode 7: Interrupt Enable 5-18
5.14 Mode 8: Interrupt Disable 5-21
5.15 Example BASIC Programs 5-22
5.16 Compiling a BASIC Program 5-22
5.17 Multiple PDMA-32's in One System 5-23
5.18 Assembly Language Programming 5-23

Section 6 APPLICATIONS INFORMATION 6-1
6.1 Typical Handshake Connection 6-1
6.2 Waveform Generation with a D/A Converter 6-3
6.3 High Speed A/D Conversion 6-4
6.4 Combined A/D and D/A Conversion using Directional Controls 6-5
6.5 Commonly Encountered Problems 6-6

Section 7 SERVICE & REPAIR 7-1
7.1 Service and Repair 7-1
7.2 Performing your own Repairs 7-1

Appendix A STORAGE OF INTEGER VARIABLES A-1

Appendix B 8254 TIMER DESCRIPTION B-1
B.1 The 8254 Programmable Interval Timer B-1
B.2 Reading and Loading the Counters B-3

Appendix C SPECIFICATIONS C-1

Appendix D COMPARISON OF PDMA-32 AND PDMA-16 D-1
D.1 General Comments D-1
D.2 Pin Connections and Functions D-1
D.3 DMA Functions D-1
D.4 I/O ports D-2
D.5 Interrupts D-3
D.6 Driver Software D-4
D.7 Other Comments D-4

Section 1

INTRODUCTION

1.1 PDMA-32 Digital D.M.A. Interface Card Description

MetraByte's PDMA-32 high speed 16 bit parallel digital interface card with D.M.A. (direct memory access) is designed to plug directly into the 16 bit (extended) expansion slots of the IBM PC/AT[1] and similar PC/AT bus based compatible machines e.g. Compaq 386 etc.

Compared to standard digital I/O boards (such as MetraByte's PIO-12) the PDMA-32 with its internal D.M.A. and interrupt control hardware is capable of sustained high speed data transfer rates which are generally difficult to accomplish with programmed I/O transfers through the main processor. Actual D.M.A. data transfer rates are computer dependent, mainly influenced by the clock rate of the 8237 D.M.A. controllers on the computer's system board. Most PC/AT bus machines (both 80286 and 80386 based) employ a 3MHz clock for these controllers and continuous transfer rates of 200,000 words/sec. or 400,000 bytes/sec. are possible. A few clones use a 4.77MHz clock for the D.M.A. controllers and are proportionately faster. Note that the D.M.A. controller is limited to making no more than 65,536 (64K) transfers to/from memory at a time. It is possible to transfer any number of bytes up to this limit as a single shot operation or constantly output or input to a block of memory using the D.M.A. controller's auto-initialize mode.

Through the use of 2 cascaded 8237 D.M.A. controllers, the PC/AT bus provides 7 D.M.A. levels. Due to the system design of PC/AT bus based computers, levels 0, 1, 2 and 3 are hardware configured to perform byte (8 bit) transfers and levels 5, 6 and 7 are hardware configured to perform word (16 bit) transfers. Level 4 is not available on the I/O bus as it is used internally to cascade one D.M.A. controller into the other. The PC/AT bus also provides 11 interrupt levels. Under software control (no switches), the PDMA-32 can be set up to perform input or output transfers on any of these 7 D.M.A. levels and generate interrupts on any of the 11 interrupt levels. The only user switch setting on the PDMA-32 is the base I/O address which can be selected on any 16 bit boundary between 100 - 3F0 Hex. Operation of multiple PDMA-32's in the same computer is simple provided they are set to different base addresses, interrupt levels and D.M.A. levels.

---

1. IBM PC and PC/AT are registered trademarks of International Business Machines Corporation

The PDMA-32 includes the following features:-

1. One 16 bit or two 8 bit digital I/O ports, A & B. The port(s) data direction can be set to input or output under software control. The port(s) may be addressed as normal I/O locations using programmed 8 or 16 bit I/O transfer instructions, or using the PC/AT's internal 8237 D.M.A. controllers, data may be directly transferred at high speed to/from the ports from/to memory. In output mode, output data can be read back from the port(s). D.M.A. transfers may be bytes (8 bit) using any of D.M.A. levels 0, 1, 2, or 3 through the A port only, or words (16 bit) using both the A and B ports combined using any of D.M.A. levels 5, 6 or 7. The A DIRECTION and B DIRECTION outputs provide information on the current direction of the ports which can simplify interface to devices capable of bi-directional data transfer. In addition 3 uncommitted auxiliary outputs AUX1-3 are available from the D.M.A. and Interrupt Control registers and can be used for control and handshake functions. When used for byte D.M.A. transfers, the B port is free for use as a standard (non-D.M.A.) port for programmed I/O. A bit in the D.M.A. control register selects whether the PDMA-32 operates as an 8 or 16 bit peripheral for I/O operations on the A & B ports regardless of the D.M.A. operating mode (byte or word) e.g. you may be set for D.M.A. word transfers but do normal I/O reads and writes as byte operations to each half of the 16 bit port. This is a useful feature since many high level languages e.g. Basic, provide byte I/O but do not support word I/O. If your programming language includes word I/O instructions, the PDMA-32 provides the capability to read/write the whole 16 bit port in a single instruction.

2. D.M.A. transfers may be initiated by an external signal (XFER REQUEST) or by an internal programmable timer. The internal timer consists of a 10MHz xtal oscillator divided through 2 sections of an 8254 counter. This can produce clock rates ranging from 2.5MHz to 0.0023Hz (about 8 pulses/hr.). The choice of external transfer request or internal clock is programmable. On receipt of a positive edge on the XFER REQUEST input, the XFER ACKNOWLEDGE output goes low and remains low until completion of the word/byte D.M.A. transfer. This takes on average about 2.5 microseconds with 8237 D.M.A. controllers operating with a 3MHz clock and worst case allowing for D.M.A. service latency does not exceed 5 microseconds. The operating D.M.A. level is selected by writing a binary code to the D.M.A. level select register. The user is required to initialize the 8237's D.M.A. controllers on the PC system board before commencing transfers (this code is provided in the driver).

3. An interrupt channel is also provided. Software control allows you to select operation on any of the eleven PC/AT bus interrupt levels 3, 4, 5, 6, 7, 9, 10, 11, 12, 14 or 15, choose between a positive or negative edge external interrupt on the INTERRUPT input pin, a periodic interrupt from the PDMA-32's internal timer or a terminal interrupt generated from the 8237 D.M.A. controllers.

4. All digital I/O connections are made through a standard 37 pin "D type" male connector which projects through the rear mounting plate.

The PDMA-32 utility software includes a callable assembly language driver for interpreted Microsoft BASIC and compiled BASIC (including Quick Basic)[2] as well as example Basic programs. The assembly language driver includes fully commented source code (PDMA32.ASM) and is modularized so it is simple to modify it or adapt it for use with other languages. The driver includes all the set up and installation software for the 8237 D.M.A. and

---

2. Quick Basic is a registered trademark of Microsoft Corporation

8259 interrupt controllers plus an example interrupt handler. All of this software is free of royalties or restrictions on making multiple copies, provided it is used in conjunction with MetraByte manufactured boards.

Typical applications for the PDMA-32 include:-

1. Interface to high speed digital peripherals.
2. Arbitrary waveform generation with an external D/A converter.
3. High speed A/D conversion with an external A/D converter.
4. High speed digital stimulus for testing and control.
5. Fast block transfer of data between computers.
6. Interrupt or D.M.A. driven background data transfers.
7. High sink current TTL digital I/O (24mA sink current).
8. Data line monitoring.
9. The PDMA-32 provides a higher performance PC/AT bus alternative to MetraByte's PDMA-16 D.M.A. I/O interface for PC/XT/AT bus computers. The PDMA-32 is pin and signal function compatible with the PDMA-16 and the PDMA-16 registers are a subset of the PDMA-32's (see Appendix D). It is easy to convert applications that were designed for the PDMA-16 to the PDMA-32 with no change in the external hardware and a substitution of the PDMA-32 software driver for the PDMA-16. The PDMA-32 offers true 16 bit D.M.A. transfers compared to the double byte transfers of the PDMA-16 and when operating in word mode is twice and fast and can move double the amount of data (64K words versus 32K words) in one transfer. It also offers a wider choice of D.M.A. and interrupt levels.

1.2 Accessories and Support

Accessories are optional and not essential for operation. They include MetraByte's SFC-37 mating solder cup 37 pin D connector and C-1800 cable / STA-U combination for users requiring screw connections. Full details of these accessories are provided in the MetraByte catalog.

MetraByte's applications engineering department is always available to provide advice and product support e.g. selecting hardware, confirming suitability for your application, choosing and debugging software and helping with any problem that you may have with any MetraByte product. There is no charge for this service, simply dial 508-880-3000 between 8 a.m. and 6 p.m. (5 p.m. Fridays) E.S.T. and ask for technical support. Telex (503989), FAX (508-880-0179) and RBBS bulletin board (508-824-8659 1200 baud) lines are available on a 24 hour, 7 day week basis.

Section 2

INSTALLATION

2.1 Backing up the Disk

The software supplied with PDMA-32 is in DOS 2.00 double sided (360K) format which can be read by all DOS versions from 2.0 and above. It is advisable to make a back up copy before using the software, although if for any reason you should lose the software, MetraByte will always provide a free replacement or you can download a copy from our bulletin board system (508-824-8659 1200 baud). The easiest way to copy the original to any other disk formatted under any other version of DOS is to insert the disk in your A drive and from DOS enter:-

COPY A:*.* B: (or other destination drive specifier)

2.2 Hardware Installation

The PDMA-32 board uses a block of 16 consecutive address locations in I/O space. Some I/O address locations will already be occupied by internal I/O and other peripheral cards, so to provide flexibility in avoiding conflict with these devices, the base I/O address can be set by the Base Address D.I.P. switch to be on an 16 bit boundary anywhere in the PC/AT decoded I/O space. This switch marked "BASE ADDRESS" is the only user setting to be made on the board before installation, all other choices, interrupt level, D.M.A. level etc. are software programmable. The PC/AT's I/O address space extends from decimal 256-1023 (Hex 100-3FF which is many times larger than is ever likely to be fully occupied. The usual I/O address assignments summarized from data in the "IBM Technical Reference Manual" are detailed on the next page.

The PDMA-32 is shipped with its Base Address DIP switch set for an I/O address of Hex 300. This is usually a good default value as it corresponds to the IBM prototype board which users seldom have installed. You may not need to alter this setting, but if you wish to check the I/O address or change it before you install the board in your computer, insert the supplied software disk in your floppy drive and enter:-

A> INSTALL

This runs a self-explanatory program (INSTALL.EXE) that will give you a pictorial view of the correct switch setting on the PDMA-32. Simply set the switch the way you see it on the screen and press <ESC> to exit back to DOS. You will also see warning messages of settings which could possibly conflict with standard IBM peripheral devices ***if you have them installed.*** If you

receive a warning for a device that is not in your computer, it can safely be ignored. These cautions apply primarily to IBM standard peripherals (although the same mapping is followed by most compatibles) and may not be totally foolproof as far as non-IBM peripherals are concerned. If your PDMA-32 does not appear to work correctly, or interferes in some way with other devices in your computer e.g. disk drives, modems etc. or your computer will not boot up, remove the PDMA-32 and try a different I/O address.

ADDRESS(Hex)	DEVICE	ADDRESS(Hex)	DEVICE
000-0FF	Internal system	380-38C	SDLC comm.
1F0-1F8	Hard disk	380-389	Binary comm. 2
200-207	Game controller	3A0-3A9	Binary comm. 1
2F0-2F7	LPT2:	3B0-3BF	Mono dsp/LPT1:
2F8-2FF	COM2:	3D0-3DF	Color graphics
300-31F	Prototype card	3F0-3F7	Diskette #1
370-377	Diskette #2	3F8-3FF	COM1:
378-37F	LPT1:]		

The table above covers the standard I/O options, but if you have other special purpose I/O peripherals e.g. tape drives, special graphics boards, prototype cards etc. they will also be sharing I/O address space. Memory addressing is separate from I/O addressing so you need not be concerned at installation with a possible conflict with any add-on memory.

Once you have set the base I/O address, make a note of its value as you will need to use it later in your programs.

To install the PDMA-32 inside your computer, start by removing the board from its protective electrostatic packaging. It is a good precaution to discharge any electrostatic charge you may have accumulated by touching the metal frame of your computer just before you plug the board in. TURN OFF THE POWER on your computer and remove the case (See IBM "Guide to Operations" for your model of computer if you are not already expert at this maneuver). Remove a vacant back plate by undoing the screw at the top and plug the PDMA-32 in and secure the back plate. The PDMA-32 will fit in any of the extended connector slots of the PC/AT bus.

Remember, always TURN OFF THE POWER whenever installing or removing any peripheral board. Failing to observe this precaution can cause costly damage to the electronics of your computer and/or the PDMA-32 board. If for any reason you later remove the PDMA-32 board, MetraByte recommends that you retain the special electrostatically shielded packaging and use it for storage.

On power up of your computer, or whenever the PC/AT bus hardware RESET line is asserted, the interrupt and D.M.A. control registers of the PDMA-32 are automatically cleared disabling all D.M.A. and interrupt functions and setting the port directions to input mode. This corresponds to a "safe" state for both internal and external hardware, so you need not be concerned about programming the PDMA-32 until it is required for use. For the correct sequences of setting up and of enabling D.M.A. and interrupts see Section 4.2.2, 4.2.3 & 5.

Section 3

UNDERSTANDING D.M.A.

3.1 What is D.M.A.?

Consider the problem of moving a large amount of data to or from an I/O device. This requirement is commonly encountered in the operation of many peripheral devices (for example disk drives) that are constantly moving large amounts of data into and out of memory. An obvious way of making the transfer would be a short program which for an input operation might read a byte from the I/O port into the accumulator (AX register) of the 8088 processor and then move the data from the AX register to a memory location to store it. In addition, we have to keep track of the memory locations where the data is going. By far the simplest way of handling this is to lay the data down in contiguous locations within a block of memory, using one of the processor's index registers to control the address. Each time a byte is transferred, the index register (usually the DI or Destination Index register) is incremented or decremented to point to the next location. A typical example assembly language program to do this might be as follows:- We could read each byte of data from memory and send it out to an I/O port using the main processor and a program loop similar to the example below:-

```
SETUP:  MOV AX,SEGMENT    ;setup segment of memory for transfers
        MOV DS,AX
        MOV DI,OFFSET     ;setup start address within segment
        MOV CX,COUNT      ;setup number of bytes to transfer
        MOV DX,IOPORT     ;DX = I/O port address

READ:   IN AL,DX          ;read byte from I/O port          (8)
        MOV [DI],AL       ;store data                       (10)
        INC DI            ;increment index                  (2)
        LOOP READ         ;continue reading until CX=0 (17)

CONT:   .....             ;yes, continue with program
```

The opposite operation of transferring data from memory to an I/O port is essentially similar. The numbers in parentheses following the READ: label are the number of processor clock cycles required to execute each instruction. On the original IBM PC which has an 8088 processor operating at a clock frequency of 4.77MHz, you would find the loop takes 8 + 10 + 2 + 17 = 37 cycles or 7.8 microseconds to transfer each byte. If you were operating with a faster processor such as a 12 MHz 80286 or 16 MHz 80386 and using exactly the code shown above, you would find transfer times are roughly inversely proportional to the clock rate e.g. a 16 MHz 80386 would take only 1.8 microseconds/byte. The 80286 and 80386 also include complex string I/O instructions (OUTSB & INSB) which are not available on the 8088. Using these instructions would be even more efficient. Note also the following:-

1. The processor is tied up 100% of the time in transferring data, it cannot be used for any other function while the transfer is in progress without interrupting the transfer.

2. The rate at which the data is transferred is controlled by the processor clock and may not correspond to the rate at which the I/O device wants to handle the data. This problem can be resolved by polling the I/O device to see if it is ready, or having the I/O device generate a hardware interrupt, but both of these solutions adds further code to the I/O routine, slowing down transfer rates even further.

3. If the processor has to handle a hardware interrupt from one device (e.g. the keyboard, COM: port or system clock) while it is involved in handling a data transfer to another device, then the delays involved may cause it to miss data, or at least will necessitate a discontinuity in the data flow.

It would be nice to have a way of transferring data without involving the processor so it can be freed up as much as possible to attend to execution of the program. It would be an added advantage if we could speed the transfer rate up compared with the example programmed transfer above, and also be able to control the rate easily. Since all we want to do is to move a byte or word directly to/from an I/O port from/to memory without any kind of intermediate processing on the way, it is better not to use the processor at all, but provide specialized hardware that will accomplish this commonly required task and tackle it faster than the processor could do it. The process of passing data to/from an I/O device directly from/to memory is known as D.M.A. (Direct Memory Access) and the hardware that controls this process is known as the D.M.A. controller, which in the case of the IBM PC/AT is handled by two 8237 D.M.A. controller chips on the system board.

3.2 The Mechanics of a D.M.A. Transfer

What happens when a device wants to transfer data to/from memory? The first step is for the device to send a signal known as the D.M.A. REQUEST (DREQ for short) to the D.M.A. controller. The processor normally has control of the computer's address and data buses as well as control signals such as the memory read/write (MEMR & MEWW) and I/O read/write (IOR & IOW) lines. To accomplish a D.M.A. transfer, control of these lines must be temporarily relinquished to the D.M.A. controller. On receipt of the DREQ, the D.M.A. controller in turn issues a HOLD REQUEST to the processor. As soon as it is able to, when it has completed any part of the instruction it is currently executing that involves a bus cycle (I/O or memory access), the processor issues a HOLD ACKNOWLEDGE signal to the D.M.A. controller, and simultaneously disconnects itself from the address, data and control buses. This process is technically known as "tri-stating" as the connections to the processor assume a third open circuit state compared to their usual binary states of 1's and 0's or highs and lows. Although the processor is a HOLD state, this does not mean it has stopped. During the hold state, the processor carries on executing parts of instructions or instructions that do not involve any external I/O action (bus cycles), when it can no longer continue, it will insert wait states until the D.M.A. controller gives the bus back.

On receipt of the HOLD ACKNOWLEDGE, the D.M.A. controller goes to work. It releases its own connections to the address and control buses from their tri-state condition, asserting a valid memory address from an internal counter and then issues a D.M.A. ACKNOWLEDGE (DACK) signal to the I/O device followed by a simultaneous IOW and MEMR for a data output, or IOR and MEMW for input. The peripheral in turn responds to the DACK and IOR or IOW signals by

placing or receiving data on the data bus effectuating a transfer directly to/from memory. On completion of the MEMR/IOW or MEMW/IOR from the D.M.A. controller, the controller removes DACK, releases HOLD REQUEST, tri-states its own address and control lines, and increments or decrements its internal address counter ready for the next transfer. The processor in turn regains control of the buses, continuing execution of the next instruction. From the assertion of DREQ to completion of the cycle takes about 2.5 - 5 microseconds, depending on the length of the instruction that the processor happens to be engaged in on receipt of the DREQ. The actual amount of time between instructions that the processor loses the bus to the D.M.A. controller is even less, about 1.7 microseconds. The effect on program execution is minimal even when transferring data at very high rates which can exceed 500,000 bytes/sec. on the PC/AT. To prevent the D.M.A. controller from "hogging" the buses if the DREQ is held constantly high, the controller always allows the processor to perform at least part of an instruction between each D.M.A. transfer, so that even operating "flat-out", D.M.A. cannot grab much more than 30% of the bus bandwidth.

In order to perform D.M.A. operations the peripheral must include hardware that generates the DREQ and responds to the DACK. The PDMA-32 includes this special hardware and this is what distinguishes it from a simple digital I/O interface that does not, such as MetraByte's model PIO-12 (Note: The PDMA-32's D.M.A. capability does not preclude it from being used as a standard I/O port for programmed I/O using IN's and OUT's). The PDMA-32 works in conjunction with the 8237 D.M.A. controller(s) which is a system component that is a part of the PC and PC/AT architecture and is essential to the operation of the PDMA-32.

It is important to appreciate that the D.M.A. controller sets the dynamics of the D.M.A. transfer, there is nothing in the peripheral I/O device that can alter the maximum speed at which the controller will handle data. There are some surprising side effects of this, in particular the IBM PC/AT which is generally 3 times faster than a standard PC or PC/XT is actually slower on D.M.A. transfers because its D.M.A. controllers clocks operate at 3MHz instead of the 4.77MHz on the PC. On the other hand it can also perform word (16 bit) transfers on its extended data bus as well as byte (8 bit) transfers on the PC compatible section of its data bus. Since word transfers amount to 2 bytes of data, the overall transfer rate in bytes per second for word transfers is higher on the PC/AT than the PC despite the slower controller clock rate but correspondingly, the single byte transfer rate is slower. The PDMA-32 is designed to take advantage of the PC/AT's word transfer capabilities as well as operate in byte transfer mode when needed.

3.3 D.M.A. Structure of the PC/AT

Although we have discussed the operation of a single device using D.M.A., it is customary to cater to the needs of several devices by providing several D.M.A. channels, each one dedicated to a particular device. The 8237 provides four separate D.M.A. channels known as Levels 0 thru 3. Correspondingly, there are 4 D.M.A. request lines, DREQ0-3, and 4 corresponding acknowledge lines, DACK0 - 3. The lines are prioritized according to a two possible protocols set by a bit in the controller command register, either fixed priority where lower D.M.A. levels have higher priority than higher levels, or rotating priority where each level takes a turn at having the highest priority. The PC BIOS sets the 8237 to operate in fixed priority mode on power up, and it is generally inadvisable to change this as it may interfere with the correct operation of the computer or other peripheral devices that use D.M.A.

The original PC and PC/XT design provided a single 8237 D.M.A. controller on the system board and the 4 D.M.A. levels were allocated to system resources as follows:-

D.M.A. Level	Function
0	Memory refresh
1	Unassigned, general I/O use
2	Floppy disk controller
3	Hard disk controller (if installed)

Apart from its uses for high speed data transfer, the D.M.A. controller includes counter hardware that cycles through the memory addresses, so as a byproduct of its design, it can also be used to refresh dynamic memory saving the cost of a separate memory refresh controller. This is what IBM chose to do in the original PC design using Level 0 to perform this function with its DREQ being driven from counter 1 of the internal 8253 timer at a 15 microsecond interval. On PC/AT memory refresh has been handled by additional hardware and does not involve the D.M.A. controller.

On PC/XT's, usually only one D.M.A. channel is available for expansion use, level 1. All levels are only capable of byte (8 bit) transfers and the controller can make up to 65,536 transfers in one operation. The PC/AT design expanded the number of D.M.A. channels by using two 8237 D.M.A. controllers (see Fig. 3.1). Since one channel (level 4) is used to cascade one controller into the other and level 0 is no longer used for memory refresh functions, the AT I/O bus provides a total of 7 channels, a considerable increase compared to the original PC bus, especially as 6 of them are uncommitted. The two controllers are hardware wired so that controller #1 accesses address lines A0 - A15 and can perform up to 65,536 byte transfers on levels 0 - 3, and controller #2 accesses address lines A1 - A16 (A0 is permanently set to zero) and can perform up to 65,536 word transfers (128 Kbytes) on even memory addresses on levels 5 - 7. The D.M.A. level assignments on the PC/AT are as follows:-

D.M.A. Level	Type	Function
0	Byte	Unassigned, general I/O use
1	Byte	Unassigned
2	Byte	Floppy disk controller
3	Byte	Unassigned
4	-	Not available (internal cascade)
5	Word	Unassigned
6	Word	Unassigned
7	Word	Unassigned

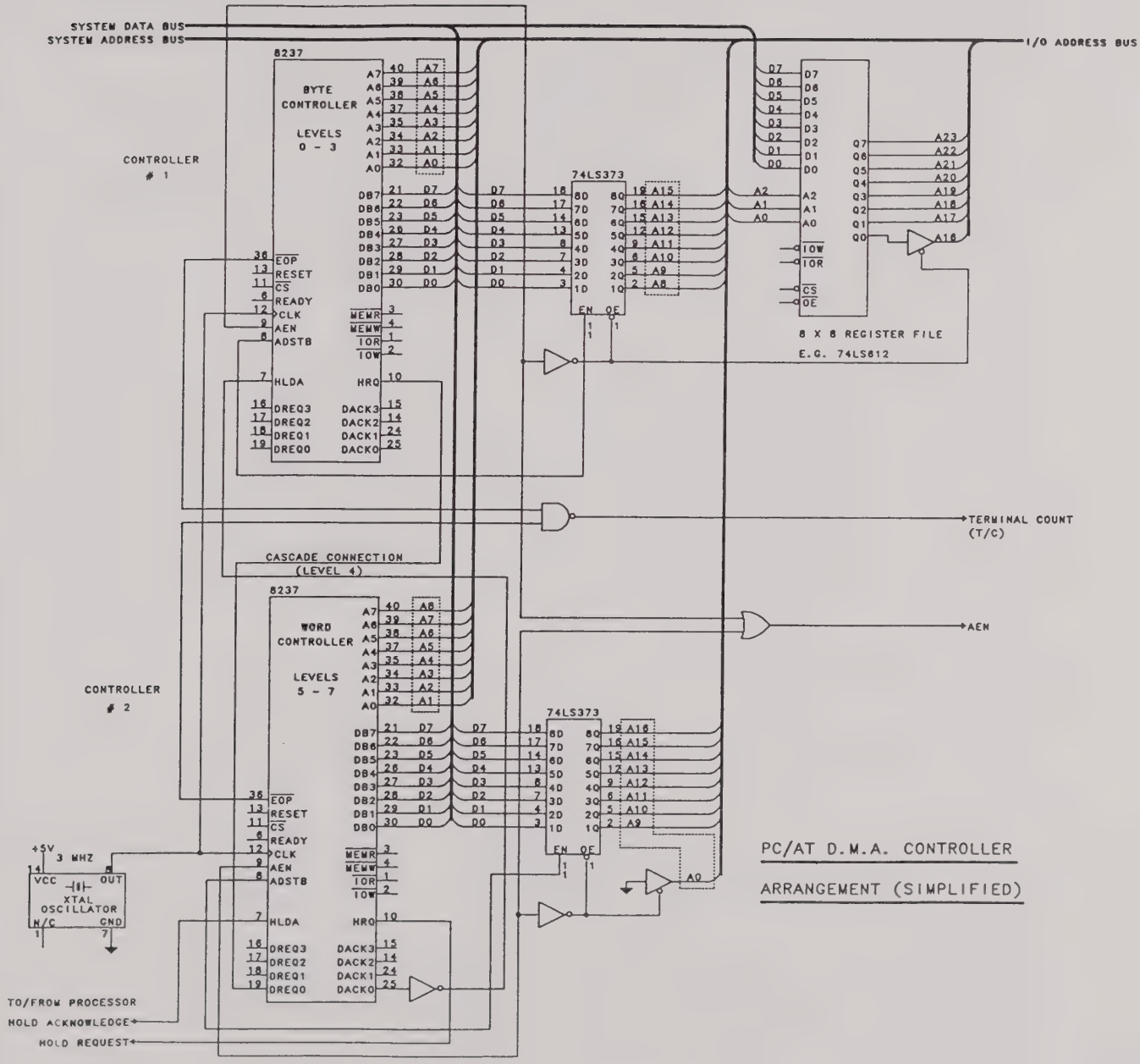


Fig. 3.1 PC/AT D.M.A. Controller Arrangement

Due to the PC/AT's internal hardware structure, you may only make byte transfers on levels 0 - 3 and word transfers on levels 5 - 7. It is important to understand that this minor restriction is imposed by the design of the PC/AT, not by the PDMA-32. If you wish, you can use D.M.A. level 2 both on the PC and PC/AT by "sharing" it with the floppy disk controller. This entails either sequentially disabling the floppy controller and enabling the PDMA-32 or vice versa. Although possible, the additional programming involved and the requirement that transfers may only be made sequentially on a shared level, coupled with the availability of so many other free levels, seldom make the programming effort involved in using level 2 worthwhile.

Note also that the PC/AT does not generally use D.M.A. for servicing the hard disk controller. This is an evolutionary fluke and has resulted from maintaining downward compatibility of the PC/AT's D.M.A. controllers structure and speed with the original PC and PC/XT. In the PC the controller is clocked with the same 4.77 MHz clock used for the main processor. The PC/AT design which started out with a 6MHz 80286 processor, and presumably for design convenience, the controllers were driven at half the main processor clock rate or 3 MHz. This meant that transfers would be a little slower than on a PC and insured backward compatibility with all the earlier PC adapters and peripheral devices. As processor clock speeds have risen, 8 - 10 -12 - 16 MHz and now 20 - 25 MHz and higher speeds, most manufacturers have played safe by maintaining the original convention of clocking the D.M.A. controllers with a separate 3 MHz clock, although there are a few exceptions to this practice on some clones (consult your technical manual). The net result of this is that a transfer takes about 60% longer on a PC/AT than a PC. If you are moving a word on each transfer, the throughput is still faster than can be achieved on a PC but much slower than can now be handled by sending data through the main processor. Hard disk controller designers are well aware of these limitations, and usually use string moves through the main processor on PC/AT's. Due to the high processor clock speeds now available, string moves are faster and more efficient than D.M.A. provided the controller card is designed with suitable buffering to handle the interruptions in data flow that will occur.

3.4 The 8237 D.M.A. Controller

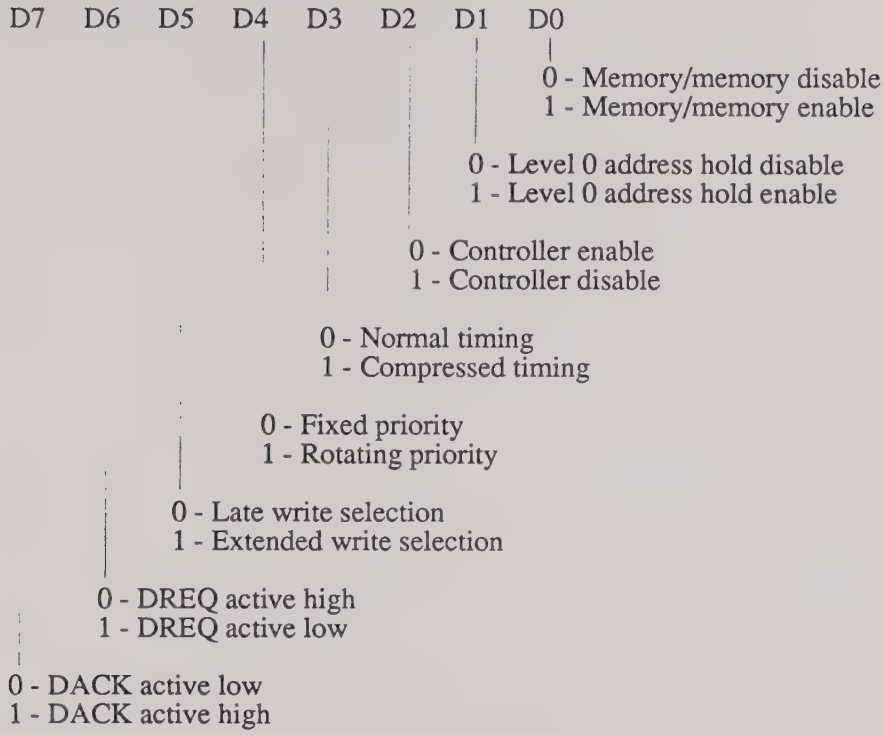
This section provides a more detailed description of the Intel 8237 D.M.A. controller used on the PC family. A full description is contained on the 8237 component data sheet available from Intel[3].

Apart from the command register, the 8237 includes several other registers using a total of 16 I/O addresses. The 8237 interfaces internally to the PC/AT data bus as an 8 bit device even though some of its internal registers are 16 bits wide. This means that all I/O must be made to the controllers using 8 bit I/O instructions. On the PC/AT, the word controller, controller 2 is connected across address lines A1 - A16 (A0 is not connected) and its I/O registers appear at every other address on even boundaries, i.e. in effect it uses 32 I/O addresses. The I/O addresses of the byte controller (#1) extend from 0 - F hex and the word controller (#2) from C0 - DF hex. A description of the registers (identical in both controllers) follows:-

---

3. Intel Corp., Lit. Dept., 3065 Bowers Avenue, Santa Clara, Ca. 95051

COMMAND REGISTER - I/O location 8H or D0H
This is an 8 bit write only register that controls overall operation of the 8237 as follows:-


D7 D6 D5 D4 D3 D2 D1 D0
0 - Memory/memory disable
1 - Memory/memory enable
0 - Level 0 address hold disable
1 - Level 0 address hold enable
0 - Controller enable
1 - Controller disable
0 - Normal timing
1 - Compressed timing
0 - Fixed priority
1 - Rotating priority
0 - Late write selection
1 - Extended write selection
0 - DREQ active high
1 - DREQ active low
0 - DACK active low
1 - DACK active high

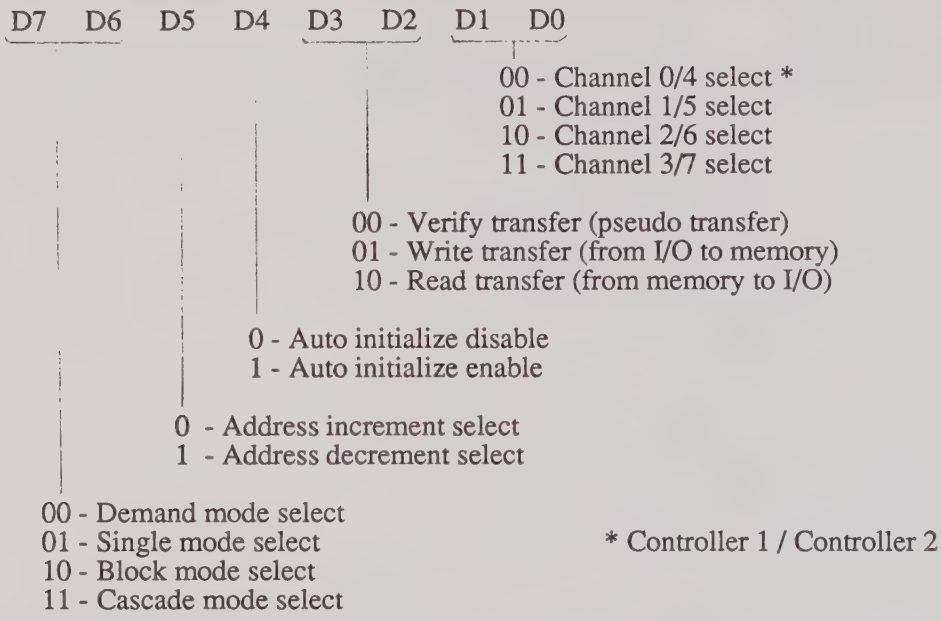
BIOS initializes the command registers of both controllers with byte 00H (zero). Since many of the bits control signal polarity and timing features that must be observed for correct operation of the internal hardware, it is inadvisable to alter BIOS's initialization of the command register byte after power up (unless you are genius who knows what he's doing!).

The key features initialized by the BIOS are:-

1. Normal timing, late write and fixed priority are selected.

2. The DREQ's are set active high, and the DACK's are set inverse, active low.

MODE REGISTER - Location 0BH or D6H
This is a group of 4 write only 8 bit registers, each one controlling the characteristics of each D.M.A. channel and selected by the lower 2 bits of the byte. The bit functions are as follows:-


D7 D6 D5 D4 D3 D2 D1 D0
00 - Channel 0/4 select *
01 - Channel 1/5 select
10 - Channel 2/6 select
11 - Channel 3/7 select
00 - Verify transfer (pseudo transfer)
01 - Write transfer (from I/O to memory)
10 - Read transfer (from memory to I/O)
0 - Auto initialize disable
1 - Auto initialize enable
0 - Address increment select
1 - Address decrement select
00 - Demand mode select
01 - Single mode select
10 - Block mode select
11 - Cascade mode select
* Controller 1 / Controller 2

A mode register is associated with each channel and unlike the command register, can safely be programmed by the user at least for the unassigned levels. Some explanation of the bit functions follows:-

D0 & D1 - The 2 lowest bits select the mode register for the channels being programmed. Make sure these are correct so that you do not inadvertently operate on the wrong channel.

D2 & D3 - Control the direction of transfer.

D4 - Controls auto-initialization (see below).

D5 - The address counter in the 8237 can count up or down from its starting value. This bit sets the count direction.

D6 & D7 - Control the type of D.M.A. transfer (see below).
00 = Demand mode select
01 = Single mode select
10 = Block mode select
11 = Cascade mode select

If bit D4 = 0, a D.M.A. transfer will start at the memory address specified in the 8237 address register and proceed for the number of bytes specified in the word count register. On reaching the specified word count, the controller automatically sets the channel mask and any further DREQ's are ignored, terminating the D.M.A. transfer. If D4 = 1, auto-initialize mode is enabled. On reaching the final word count, the controller re-initializes the address counter with its starting address, so that in this mode D.M.A. proceeds continuously to a memory buffer area of location specified by the starting address, and length specified by the word count. On reaching the end of the buffer, transfers continue at the beginning of the buffer. In effect the buffer is circular in auto-initialize mode. This can be useful when outputting data of a cyclic nature such as driving a D/A converter to generate a periodic waveform or when streaming data to/from disk.

D6 and D7 control the D.M.A. transfer modes. BIOS initializes D6 and D7 on every channel to select the single transfer mode. In this mode, each DREQ will result in a single byte transfer of data followed by return of bus control to the main processor. The PDMA-32 has been designed for operation of the D.M.A. controller in this mode and this is the only valid selection. The block and demand modes are almost similar to each other. In block mode, a single DREQ will make the D.M.A. controller perform multiple transfers one after another until the final word count is reached. Once the D.M.A. process becomes active, the DREQ can be taken low without affecting continuation of the D.M.A. block transfer i.e. the DREQ is ignored once transfers start. Demand mode is similar except that the transfers can be halted at any time during the block transfer by taking DREQ low. Transfers will continue from the point where they stopped on taking DREQ high. Both of these transfer modes can block the processor's use of the bus for extended times and may hold off service of other D.M.A. channels while they are active. Since the use of block and demand modes was not allowed for in the original PC/AT design, it is inadvisable to try using them as they may produce system problems and in any case the PDMA-32 has not been designed to support them. Cascade mode allows connection of another 8237 hold request and hold acknowledge signals to the DREQ and DACK of the first D.M.A. controller. On the PC/AT, level 4 of the primary controller is used for cascade expansion in this way and is programmed by the BIOS accordingly.

MASK REGISTER - I/O locations 0AH/0FH or D4H/DEH
Each channel has associated with it a mask bit which can be set to disable the incoming DREQ. A mask bit is also set when not in the auto initialize mode by the terminal count, T/C, produced at the end of the D.M.A. transfer. The mask register including all the mask bits for all the channels can be written to as a whole through the port at I/O location 0FH (controller #1) or DEH (controller #2). Apart from initializing the D.M.A. controller which the BIOS performs on power up, it is best to avoid this method of setting and resetting mask bits, since you may disturb the mask bit on another channel inadvertently as the mask register is write only and it is impossible to read its current contents.

The I/O locations at address 0AH (controller #1) or D4H (controller #2) provide a better way of setting and clearing a specific channel mask register bit without disturbing the mask bits of the other D.M.A. channels. The protocol for doing this is as follows:-

D7	D6	D5	D4	D3	D2	D1	D0

Don't care

00 - Select level 0/4 mask bit
01 - Select level 1/5 mask bit
10 - Select level 2/6 mask bit
11 - Select level 3/7 mask bit

0 - Clear mask bit (enable D.M.A.)
1 - Set mask bit (disable D.M.A.)

Before performing any setup of the D.M.A. controller, set the mask bit of the level being programmed to disable any spurious D.M.A. transfer that might otherwise occur.

ADDRESS REGISTERS - Location (2 x Level #) or C0H + (4 x Level (# - 4)
There are four sixteen bit read/write address registers in each controller, one associated with each D.M.A. level. The address register should be loaded with the offset of the starting address of the memory buffer area reserved for D.M.A. Since these are 16 bit registers, they are loaded by 2 sequential outputs to the same I/O port e.g. for Level 3 (example in BASIC):-

```
OUT 6, Low byte of address
OUT 6, High byte of address
```

If you are programming in assembly language, ***you must insert a delay between back to back I/O instructions when using 80286/386 processors***. This is to allow sufficient recovery time for the I/O bus (see IBM Technical Reference Manual). Failure to insert a delay will lead to erroneous reads and writes and erratic results. A short jump provides an adequate delay. Also, since the 8237 uses an internal byte pointer flip-flop to sequence the low byte / high byte reads and writes, it is important this is cleared prior to a read or write operation and is not altered by an interrupt during execution of your code. The byte pointer flip-flop is cleared by writing any data to I/O address 0CH for byte controller #1 or D8H for word controller #2. The following assembly language example shows loading the address register of level 5 including all the correct precautions and steps:-

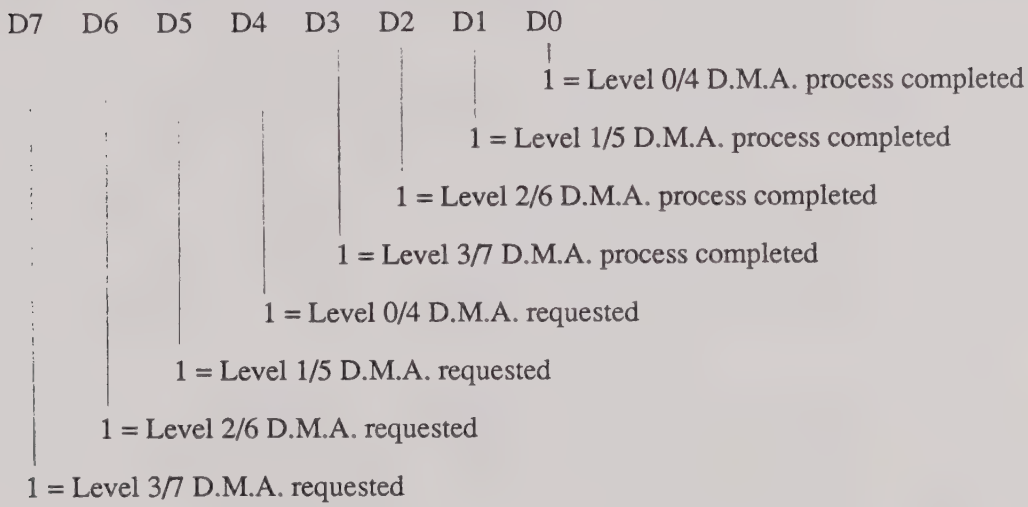
```
MOV AX,DATA     ;get address data to load in AX
CLI             ;disable interrupts
OUT 0D8H,AL     ;clear byte pointer flip-flop
JMP $+2         ;I/O delay (recommended)
OUT 0C4H,AL     ;write low byte
MOV AL,AH       ;transfer high byte to AL
JMP $+2         ;I/O delay (required)
OUT 0C4H,AL     ;write high byte
STI             ;re-enable interrupts
```

The address read back from the same location is the current address i.e. the starting address plus or minus the number of D.M.A. transfers depending on whether increment or decrement is selected by the mode register.

TRANSFER COUNT REGISTERS - Location (2 x Level #)+1 or C0H+(4 x Level (#-4)+2)
There are four sixteen bit transfer count registers (called word count on the Intel data sheet) that control the number of D.M.A. transfers. These registers are loaded with the number of transfers required in an operation minus one. The registers decrement through zero and on reaching FFFFH generate a terminal count, T/C, on the expansion bus. In the non-auto initialize mode further D.M.A. automatically masks off and further transfers cease, whereas in the auto initialize

mode both the transfer count and address registers are automatically reloaded with their initial values and D.M.A. proceeds continuously. The combination of the T/C (terminal count) I/O bus signal and the DACK for the selected D.M.A. level can provide an interrupt to signal the end of the D.M.A. transfer. The PDMA-32 can generate this type of interrupt on any level if required. Other methods of determining if the D.M.A. transfer has ended is to poll (read) the controller status register (I/O address 8H controller #1 or D0H controller #2) or read the transfer count register until it has reached FFFFH. This latter method also provides data on the number of D.M.A. transfers that have taken place and is the method used by mode 2 of the PDMA32.BIN driver.

STATUS REGISTER - Location 8H or D0H
The D.M.A. controller status register provides information on whether a channel is active and if a D.M.A. transfer has finished:-


D7 D6 D5 D4 D3 D2 D1 D0
1 = Level 0/4 D.M.A. process completed
1 = Level 1/5 D.M.A. process completed
1 = Level 2/6 D.M.A. process completed
1 = Level 3/7 D.M.A. process completed
1 = Level 0/4 D.M.A. requested
1 = Level 1/5 D.M.A. requested
1 = Level 2/6 D.M.A. requested
1 = Level 3/7 D.M.A. requested

BYTE POINTER FLIP-FLOP - Location 0CH or D8H
When reading or writing any of the 16 bit internal registers of the 8237 (the address or transfer count registers) that involve double 8 bit sequential I/O operations to the same I/O address, the 8237 uses an internal sequencing flip-flop to select the low and high byte portions of the register. It is advisable to clear this flip-flop before one of these double byte read or write operations so that data is sequenced in the correct order. Writing any data to I/O addresses 0CH (controller #1) or D8H (controller #2) clears the byte pointer flip-flop.

3.5 The D.M.A. Page Registers

The 8237 D.M.A. controller is an older peripheral device that was designed to work with a 16 bit address bus which these days is only adequate to control a small portion of the total addressable memory. The original PC used a 20 bit address bus for the 1 Mbyte of RAM and ROM memory allowed in the PC architecture, the PC/AT provides a 24 bit address bus to address a physical maximum of 16 Mbytes of memory. It is therefore only possible to perform D.M.A. within a 16 bit addressable area of memory at a time. This area is termed a page and for the byte controller amounts to 64Kbytes of memory, for the word controller it is 128 Kbytes as A0 the least

significant address byte is always zero. The page address is provided by a set of 8 bit registers (4 bit on the original PC) external to the 8237 D.M.A. controllers. There is a register for each D.M.A. level except internal level 4. The byte controller #1 provides bits A0 - A15 of the address and the page register provides the remaining address bits, A16 - A23. The word controller #2 provides bits A1 - A16 of the address, the page register provides bits A17 - A23, the least significant bit of the word page registers is ignored and the least significant bit of the address bus (A0) is set to zero so that word transfers for levels 5 - 7 may only be made on even address boundaries.

It is important to understand that although the page register supplements the D.M.A. controller to provide the full 24 bit address, there is no way that the address generating counter in the 8237 can propagate a carry/borrow into the page registers. This leads to an important programming precaution, always be sure that your transfer count plus your base memory address will not take you across a page boundary. If it does, since the page address does not increment, you will get a wrap around to the beginning of the page which more often than not will overwrite some essential program memory and crash the computer!

The 8 bit page registers may be read/write or write only registers depending on the type and design of your computer. The page register locations and functions are as follows:-

I/O LOCATION	FUNCTION
87 Hex	Level 0 page register
83 Hex	Level 1 page register
81 Hex	Level 2 page register
82 Hex	Level 3 page register
8B Hex	Level 5 page register
89 Hex	Level 6 page register
8A Hex	Level 7 page register

Note that there is no simple relationship between the page register I/O address and its associated D.M.A. level, this just makes programming a little more of a challenge!

3.6 Setting up a D.M.A. Transfer

Setup of a D.M.A. transfer entails several steps which must be performed in the correct sequence and requires a good understanding of the action of the hardware. Mode 1 of the PDMA32.BIN driver performs this setup and avoids a lot of programming, the source listing of mode 1 (see PDMA32.ASM) provides an example of the controller setup. If you do not want to use the driver and wish to write your own setup program you should proceed in the following sequence:-

1. Disable any active D.M.A. on the chosen level by setting the appropriate mask register bit or alternatively wait for the transfer to finish.

2. Write to the appropriate mode register of the appropriate 8237 D.M.A. controller to set up the transfer characteristics.

3. Clear the byte pointer flip-flop and disable interrupts (clearing interrupts in a high level language may not be possible - you can take your chances or use the driver).

4. Load the address and byte count registers with appropriate data.

5. Re-enable interrupts.

6. Write the page register to select the D.M.A. page.

7. If you intend to make use of a terminal count interrupt to signal the end of D.M.A. then install your interrupt handler (set interrupt vector and 8259 controller(s)).

8. Set the PDMA-32 to operate on the D.M.A. and interrupt levels selected by writing the level numbers to the PDMA-32 D.M.A. and interrupt level select registers and then enable D.M.A. and interrupts from the PDMA-32 using its D.M.A. and interrupt control registers.

9. If you are using the PDMA-32's on-board timer to generate D.M.A. transfer requests, it is a good idea to reload it so that the first D.M.A. transfer will occur at a fixed delay from this point. An excellent technique is to load the timer with the largest division ratio possible which will produce a pulse every 7 minutes - in effect disabling the timer. The timer can later be set to its desired speed after all the D.M.A. and interrupt setup is completed.

10. Enable 8259 interrupt controller(s) mask register (if using interrupts)

11. Last step, enable D.M.A. controller mask register.

12. If you are using the on-board timer to pace D.M.A. transfers, now is the point to bring it up to correct speed (see Step 9).

The main pitfall to avoid when doing the D.M.A. setup is enabling the 8237 D.M.A. controller before D.M.A. is enabled on the PDMA-32 through bit 7 of the D.M.A. control register. Until the PDMA-32 is enabled, the DREQ line is tri-state or high impedance and picks up a lot of crosstalk from other lines on the expansion bus. If the 8237 mask register bit is enabled, many spurious rapid D.M.A. transfers will be generated, and it is not unusual to find that the 8237 signals that transfers are ended before the PDMA-32 is later enabled! Always enable the PDMA-32 first followed by enabling the 8237 controller.

Another detail that is not made very obvious in the 8237 data sheet is that while a channel is masked off, it will store a DREQ. As soon as the channel mask bit is cleared (enabled) this stored request gets serviced and you often get an undesired transfer immediately on enabling the controller. Since you usually do not know whether there is going to be a stored request pending, in some applications, you may have to enable the controller and reload the address and transfer count registers with the desired values and then start up your D.M.A. transfers. Generally if the DREQ bus line input has been left in an open circuit or tri-state condition, bus crosstalk will have induced enough noise into the line to generate a pending D.M.A. request. This is usually the condition on making the first set of transfers with the PDMA-32.

3.7 D.M.A. Transfer Timing

On receipt of a D.M.A. request (DREQ) from a peripheral, the 8237 D.M.A. controller immediately sends a HOLD request to the 80286/386 processor. The bus interface unit responds to the HOLD as soon as possible according to the following rules:-

1. The response is close to immediate if no bus cycle (transfer to or from memory or I/O) is in progress. This time can be as little as 0.1 microseconds.

2. If a bus cycle is in progress, it will be completed before hold acknowledge is enabled. In this case a processor clock speed dependent delay of up to 200-600nS (nanoseconds) may take place. Memory wait states will add to this delay.

3. If a bus cycle involving a word (2 byte) transfer to an odd byte memory boundary is taking place, both bus cycles will be completed before hold acknowledge is enabled. In this case the delay can extend beyond a microsecond.

4. If the hold request occurs at the beginning of an interrupt acknowledge sequence then hold acknowledge can be delayed somewhat in excess of 13 processor clock periods, 1 - 3 microseconds.

5. If an instruction is executed with a LOCK prefix, the bus interface unit will insure that the whole following instruction will be executed before issuing a hold acknowledge. In the case of instructions that require several memory references during execution e.g. ADD [BX],CX this could add many microseconds of delay.

In general condition 5 is unlikely to arise, since the LOCK prefix is used by programmers to insure total instruction execution in multi-processor systems and the PC/AT is a single processor system. There is therefore no reason for a programmer to use the LOCK prefix in programs for the PC, though of course there is nothing to prevent its use.

Once the D.M.A. transfer is under way, the D.M.A. controller will take 5 clock cycles or 1.7 microseconds (3 MHz clock) to effect the transfer. Assuming that instructions are not LOCKed, the worst case transfer timing would appear to be condition 4 plus the D.M.A. cycle time, around 4 microseconds. This neglects the effect of higher level D.M.A. channels which if active and requesting service can extend the delay. The technical term for this service delay is called D.M.A. latency.

A practical method of measuring the latency is to set the PDMA-32 up in auto initialize mode with the D.M.A. requests driven by the timer which should be run at a pulse rate of a few KHz. You can synchronize a scope from the timer pulse output (pin 22) the positive edge of which generates the DREQ. If you examine the XFER. ACK. (pin 3), it goes low on the start or positive edge of the DREQ and rises back to a high state on completion of the D.M.A. transfer at the end of DACK. The trailing edge will appear as a blur on the scope, and the blur extends anywhere from 2.5 to somewhat less than 5 microseconds from the start of the pulse. This corresponds to the best and worst case D.M.A. transfer times. You will notice that the majority of transfers occur in 2 to 3 microseconds.

There are a number of conclusions that can be drawn from these numbers. If you are performing clocked D.M.A. transfers i.e. regulated by the timer, it will limit transfer speeds to the worst case delay time corresponding to about 200,000 transfers per second. Note this applies whether you

are transferring bytes (levels 0 - 3) or words (levels 5 - 7) so the corresponding data rates are 200,000 bytes/sec or 200,000 words/sec (400,000 bytes/sec.). If you are able to use the XFER. REQ. and XFER. ACK. to handshake with the peripheral and the peripheral can provide data as fast as it can be transferred, you will no longer be limited by the worst case transfer time and you can expect speeds of up to 350,000 transfers/sec.

3.8 Using Interrupts with D.M.A.

The PDMA-32 can generate a hardware interrupt on any of the 11 PC/AT levels available on the bus. These are levels from 3, 4, 5, 6 ,7, 9, 10, 11, 12, 14 & 15. The interrupt can be generated from one of the following three sources:-

1. An external positive or negative edge input on the INTERRUPT IN (pin 1).

2. A periodic interrupt from the timer.

3. A terminal count (T/C) from the D.M.A. controller

This provides several options in programming. The D.M.A. terminal count interrupt can be used to signal the end of a D.M.A. transfer and start up another operation e.g. another D.M.A. transfer or move data from the D.M.A. buffer to be processed etc. The external interrupt can be supplied by an external device that requires D.M.A. service, or to signal that it has transferred all its data etc. The periodic interrupt can be used to perform some repetitive operation like sending a block of data to a peripheral.

Since there are many possible actions that a user may require from an interrupt service routine, it is impossible to provide a ready made all purpose routine. The PDMA32.BIN driver does include a sample interrupt service handler (labelled INTH:) that can be installed and enabled using mode 7 and disabled with mode 8. This particular routine generates an audible "beep" every time it is invoked (run INT.BAS for a sample). A programmer can readily modify the handler to his own needs and re-assemble the PDMA32.ASM source. Note that most high level languages, including BASIC, do not provide a means of writing an interrupt service routine and it has to be done through the driver as described. Modes 7 and 8 provide examples of performing the set up of the PC/AT's 8259 interrupt controllers on any level which is fairly complex. One further precaution, when you write interrupt handlers avoid using DOS calls within them, since most DOS calls are not re-entrant. Your program can be in the middle of using a DOS call when it is interrupted and if your handler happens to use the same or similar DOS call, things get altered in strange ways and programs crash! This mistake can lead to difficult to diagnose, randomly appearing bugs, so be aware of it.

3.9 Determining Status of a D.M.A. Transfer

Once a D.M.A. transfer is under way, it is often necessary to find out how many transfers have taken place or whether the D.M.A. transfer has completed. This can be determined by reading the appropriate byte count register of the 8237 D.M.A. controller e.g.:-

```
10 XL% = INP(7)     'low byte for level 3
20 XH% = INP(7)     'high byte for level 3
```

```
30 B = 256 * XH% + XL%  'B = number of transfers left to complete
```

Alternatively, mode 2 of the PDMA32.BIN driver will perform a similar function, returning the number of bytes or words transferred according to the setup of mode 1.

Section 4

REGISTER STRUCTURE & CONNECTIONS

4.1 I/O Map

The PDMA-32 uses 16 consecutive locations in I/O address space (this does not interfere with memory addressing in any way). The BASE ADDRESS is set by a 5 position DIP switch and can be on any 16 bit boundary in the PC/AT's decoded I/O address range of 100 to 3F0 hex. The addresses are mapped as follows:-

ADDRESS	FUNCTION	TYPE
BASE + 0	A port	R/W
+ 1	B port	R/W
+ 2	D.M.A. control	R/W
+ 3	Interrupt control	R/W
+ 4	Counter 0	R/W
+ 5	Counter 1	R/W
+ 6	Counter 2	R/W
+ 7	Counter Control Counter Status	W R
+ 8	D.M.A. level	R/W
+ 9	Interrupt level	R/W
+ A	Interrupt status	R
+ B-F	Not used	-

(R = read, W = write)

Note that addresses Base + 4 thru 7 correspond to the 8254 timer.

4.2 Register Functions

4.2.1 Ports A & B

Ports A and B are the main digital I/O ports. They are each 8 bits wide and can be used individually or combined as one 16 bit port for both programmed I/O and D.M.A. I/O. Each port has a data direction output associated with it (A DIR & B DIR). The data directions are selected by bits D0 and D1 of the D.M.A. control register and on power up are always reset to the input mode. The following operating modes selected by bit D2 of the D.M.A. control register and the level programmed in the D.M.A. level register are possible:-

Bit D2	D.M.A. Level	Port I/O	A & B direction
0 - byte	0 - 3 (byte)	Byte for I/O Byte for D.M.A.	A & B independent
1 - word	0 - 3 (byte)	Word for I/O Byte (PA) for D.M.A.	A controls both ports B irrelevant
0 - byte	5 - 7 (word)	Byte for I/O Word for D.M.A.	A controls both ports B irrelevant
1 - word	5 - 7 (word)	Word for I/O Word for D.M.A.	A controls both ports B irrelevant

Byte wide (8 bit) D.M.A. operations can only be made through port A. In this case port B can is available independent of port A for input or output using programmed I/O.

Word wide (16 bit) D.M.A. operations are made through both ports A and B combined. In word mode, port A provides the least significant byte and port B provides the most significant byte of data. Of necessity, the data direction of both ports has to be identical if a word transfer D.M.A. level 5 - 7 is selected, and the B direction set by bit D1 of the D.M.A. control register is ignored both for D.M.A. and programmed I/O.

For normal programmed (non-D.M.A.) I/O you have a choice of accessing the ports as 2 separate byte wide ports and using byte oriented I/O instructions e.g. IN AL,DX or OUT DX,AL (assembly) or INP(---) or OUT ---, --- in BASIC to I/O addresses BASE or BASE + 1 or configuring them as a single 16 bit word wide port (A & B ports combined) and using word I/O instructions e.g. IN AX,DX or OUT DX,AX (assembly) to I/O address BASE. The possible combinations are delineated in the table above. Note that many high level languages e.g. Basic do not have word I/O instructions, so either use the PDMA32.BIN driver or set the ports for byte oriented access and use Basic's INP"s and OUT's.

The port data format for 8 bit operations is:-

			D7	D6	D5	D4	D3	D2	D1	D0
BASE	+	0	A7	A6	A5	A4	A3	A2	A1	A0
	+	1	B7	B6	B5	B4	B3	B2	B1	B0

For 16 bit operations, the data format is:-

			D15	D14	D13	D12	D11	D10	D9	D8
BASE	+	0	B7	B6	B5	B4	B3	B2	B1	B0
			D7	D6	D5	D4	D3	D2	D1	D0
			A7	A6	A5	A4	A3	A2	A1	A0

Notes:-

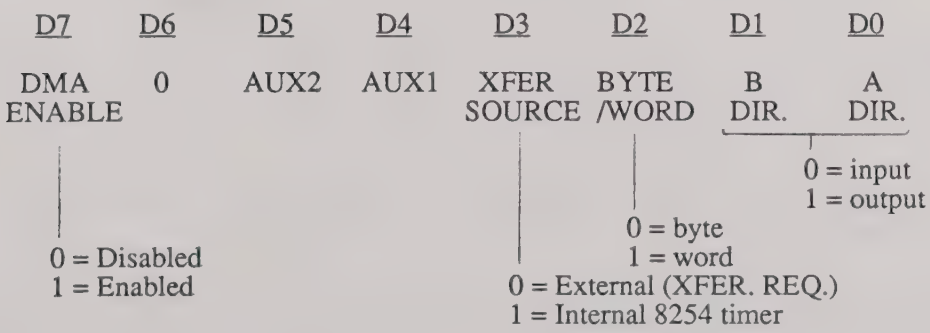
1. Both ports are automatically set up in the input direction on power up of the computer (when hardware reset bus line is active). Their data and configuration will not be changed by a soft boot (Ctrl-Alt-Del sequence).

2. In input mode each line of port A presents 1 74LS load to the driving source and each line of port B presents 2 74LS loads to the source. In output mode, outputs of both ports will sink 24mA and are 74LS and standard TTL compatible. For driving CMOS inputs, 330 ohm pull up resistors to +5v are recommended.

3. In output mode, data can be read back from either port. This data corresponds to the actual data on the output pins and may not correspond to data written if an output line is shorted or faulty.

4.2.2 D.M.A. Control Register

The D.M.A. control register is an 8 bit read/write register located at I/O address BASE + 2. The D.M.A. control register bits have the following functions:-



Notes:-

1. The D.M.A. control register is cleared on power up (reset) of the computer, thus disabling D.M.A. and setting ports A & B as inputs.

2. If word mode is selected for regular I/O (D2 = 1) then the B direction bit D1 is ignored. Both ports operate as a single 16 bit port with the direction set by the A direction, bit D0.

3. To avoid spurious transfers, the DMA ENABLE bit should be set before enabling the mask register of the 8237 D.M.A. controller in the computer.

4. The desired D.M.A. level is selected before enabling D.M.A. by writing it to the D.M.A. level select register at I/O address BASE + 8. This should be done before enabling D.M.A.

5. When DMA ENABLE = 0, the associated D.M.A. I/O bus signal is tri-stated and may be shared by other devices.

6. The AUX1 & 2 bits correspond to unused bits. For convenience these have been brought out to provide additional outputs on the rear connector.

7. Bit D6 will always read back as 0 regardless of what is written to it.

4.2.3 Interrupt Control Register

The interrupt control register is an 8 bit read/write register located at I/O address BASE + 3. The bits have the following functions:-

D7	D6	D5	D4	D3	D2	D1	D0
INT ENABLE	0	0	0	AUX3	INT	SOURCE	SLOPE

D0: 0 = + edge, 1 = - edge

D1: 00 = External input, 01 = 8237 terminal, 10 = 8254 timer, 11 = 8237 terminal

D7: 0 = Disabled, 1 = Enabled

Notes:-

1. The interrupt control register is cleared on power up (reset) of the computer, thus disabling interrupts.

2. The interrupt level is selected by writing the required level to the Interrupt Level Select register at I/O address BASE + 9. This should be done before enabling interrupts.

3. The INT ENABLE bit should be set before enabling the 8259 interrupt controller mask register to avoid generation of spurious interrupts.

4. When INT ENABLE = 0, the associated I/O bus interrupt line is tri-stated and available for use by other devices.

5. The AUX3 bit corresponds to an unused bit. For convenience it has been brought out to provide an additional output on the rear connector.

6. Bits D4 through D6 will always read back as 0 regardless of what is written to them.

4.2.4 8254 Timer

I/O locations BASE + 4 thru 7 correspond to the 8254 timer. A full description of the 8254 capabilities and programming is provided in the Intel 8254 data sheet and a partial description, adequate for programming, is supplied in Appendix B. Its use in the PDMA-32 is purely for periodic timing pulse generation and its connection is shown below. Note that counter 2 is not hardware accessible to the user although it may be loaded and read for timing purposes.

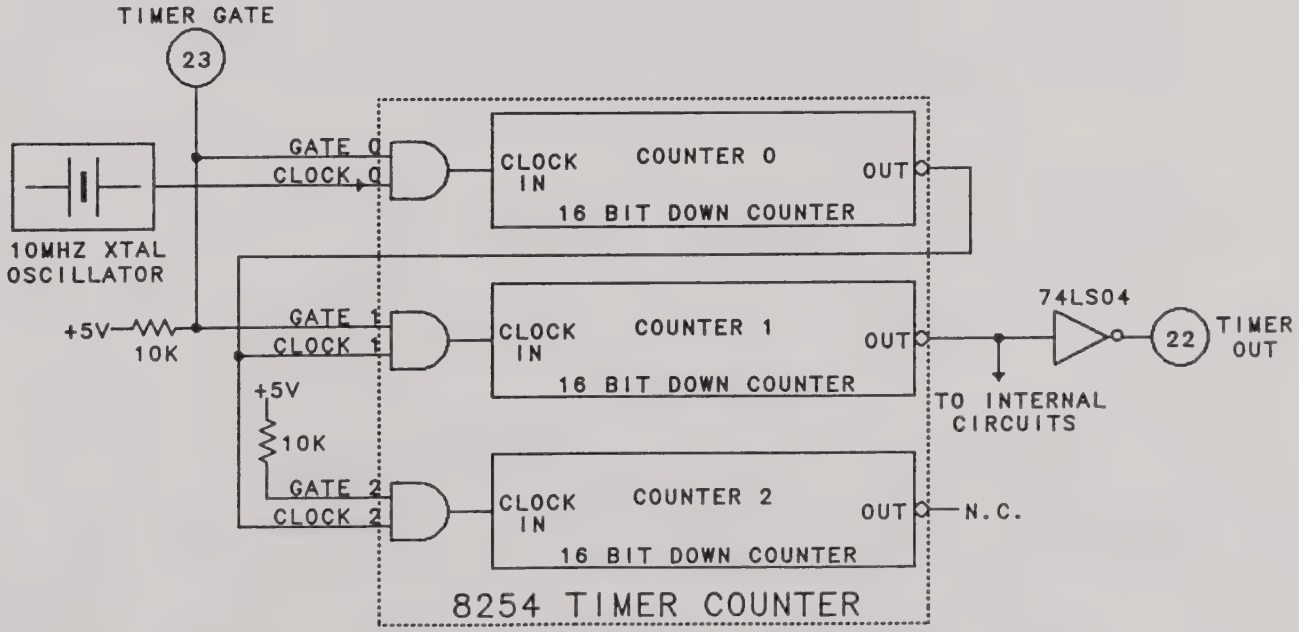


Fig. 4.1 Diagram of 8254 connection in PDMA-32

4.2.5 D.M.A. Level Select Register

This is a read/write register at I/O address BASE + 8. The state of the bits are arbitrary after power up.

D7	D6	D5	D4	D3	D2	D1	D0
x	x	x	x	x	L2	L1	L0

x = don't care (you can read & write these bits but they have no effect)

000 - Level 0 (byte)
001 - Level 1 (byte)
010 - Level 2 (byte)
011 - Level 3 (byte)
100 - Illegal (inactive)
101 - Level 5 (word)
110 - Level 6 (word)
111 - Level 7 (word)

Note that D.M.A. level 2 can be selected but if used will normally interfere with the operation of the floppy diskette. Writing 04H to I/O port 3F2H will temporarily disable the DREQ2 from the floppy disk adapter. You can then operate the PDMA-32 on level 2, when you are finished, disable the PDMA-32 D.M.A. and DOS will automatically re-enable the floppy diskette adapter on the first disk access. Obviously, if you are operating on level 2, avoid any concurrent diskette and PDMA-32 activity with both devices enabled.

4.2.6 Interrupt Level Register

This is a read/write register at I/O address BASE + 9. The state of the bits is arbitrary after power up. Note that the Interrupt Control register is cleared on power up (hardware reset).

D7	D6	D5	D4	D3	D2	D1	D0
x	x	x	x	I3	I2	I1	I0

x = don't care
(you can read & write these bits but they have no effect)

0000 - Illegal (inactive)
0001 - Illegal (inactive)
0010 - Illegal (inactive)
0011 - Level 3
0100 - Level 4
0101 - Level 5
0110 - Level 6
0111 - Level 7
1000 - Illegal (inactive)
1001 - Level 9
1010 - Level 10
1011 - Level 11
1100 - Level 12
1101 - Illegal (inactive)
1110 - Level 14
1111 - Level 15

Interrupt priorities highest to lowest are 9, 10, 11, 12, 14, 15, 3, 4, 5, 6, 7. There is no interrupt level 2 on the PC/AT, it is redirected through interrupt level 9. According to the IBM Technical Reference Manual for the PC/AT, levels 9, 10, 11, 12 & 15 are designated "Reserved" i.e. potentially free for user applications.

4.2.7 Interrupt Status Register

This is a read only register at BASE + 0AH. It returns a single bit corresponding to the state of the PDMA-32's internal interrupt request. Reading the interrupt status register will clear the interrupt request and re-enable interrupts from the PDMA-32, this should normally be done at some point in your interrupt handler and prior to setting up and enabling the 8259 interrupt controller(s).

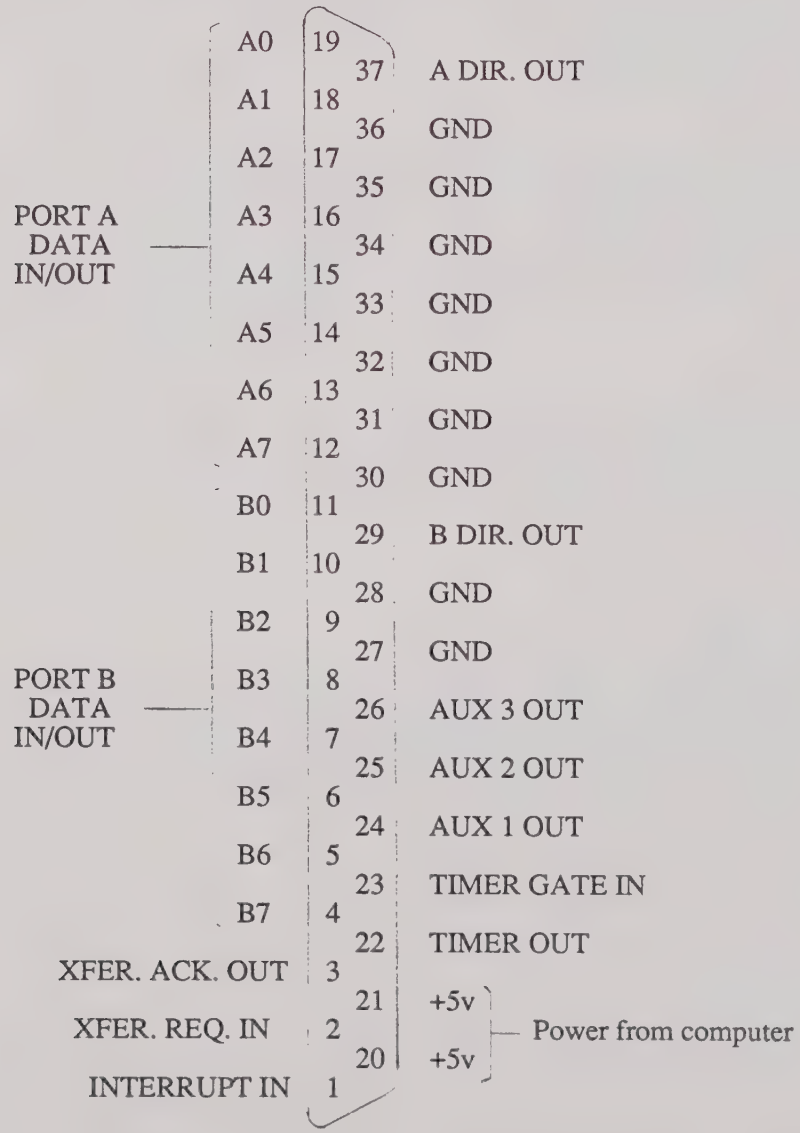
D7	D6	D5	D4	D3	D2	D1	D0
IRQ	0	0	0	0	0	0	0

1 = interrupt active
0 = no interrupt

4.3 Connector Pin Assignment

A standard 37 pin D type male connector is used for all I/O which is through the rear mounting plate. The mating connector is a 37 pin D type female such as ITT/Cannon #DC-37S for soldered connections (MetraByte accessory #SFC-37) or AMP #745242-1 for flat insulation displacement cable. If you wish to access connections with screw connectors, use MetraByte STA-U universal screw connector board with either C-1800 unshielded or S-1800 shielded cable accessories.

REAR VIEW OF PDMA-32 I/O CONNECTOR



4.4 I/O Signal Functions

A0 - A7	:	Port A data (input/output)
B0 - B7	:	Port B data (input/output)
A DIR	:	Port A direction (output) 0 = Port A input 1 = Port A output
B DIR	:	Port B direction (output) 0 = Port B input 1 = Port B output
XFER. REQ.	:	A positive edge on this input initiates a D.M.A. transfer with D.M.A. enabled and D3 of the D.M.A. control register = 0.
XFER. ACK.	:	On receipt of a XFER. REQ. the XFER. ACK. goes low. After the 8237 D.M.A. has finished the byte or word transfer, XFER. ACK. returns high indicating that valid data is on the port(s) in output mode, or that data has been transferred in input mode.
TIMER OUT	:	This is a positive pulse or square wave output from the timer.
TIMER GATE	:	When low, this input inhibits (or holds) external and internal pulses from the timer. The TIMER GATE has an internal 10K pullup resistor to +5v, if not used this input can be left disconnected.
INTERRUPT	:	This is a positive or negative edge triggered interrupt input. The slope of the trigger edge is selected by bit D0 of the interrupt control register.
AUX1 - 3	:	These are general purpose control output lines corresponding to spare bits in the D.M.A. and Interrupt control registers.
+5v	:	This is the +5v logic supply from the computer. Provided that you are sure you can avoid short circuits, overloads and application of external voltages i.e. any type of abuse which might damage the computer, it may be used to power external peripherals. Be sure to observe available power limits, in any case do not exceed 2 amps due to connector limitations - see Technical Reference Manual.
GND	:	Logic and power return ground.

Section 5

PROGRAMMING

5.1 Programming PDMA-32

This section provides information to programmers who wish to use the MetraByte utility software supplied with PDMA-32.

At the lowest level, PDMA-32 can be programmed using I/O input and output instructions such as BASIC's INP (port) and OUT port,data functions or assembly language IN AL,DX and OUT DX,AL. Most other languages (with the exception of Fortran) have equivalent instructions. For Microsoft Fortran, MetraByte can supply a library (MetraByte FOR-16) that includes both I/O and memory access extensions.

Programming directly with I/O commands involves formatting data and dealing with absolute I/O addresses. Although not demanding, this usually requires many lines of code and necessitates a fairly good understanding of the register functions, data format and method of operation of the PDMA-32. Also many languages, such as BASIC, do not support the writing of interrupt routines, and this is a major obstacle in many applications as interrupts can often be effectively utilized in the operation of the PDMA-32. To simplify program generation in BASIC, a special I/O driver routine "PDMA32.BIN" is included in the PDMA-32 software package.

The object file of the driver, PDMA32.OBJ is included for linking when compiling your BASICA program using the IBM Basic Compiler or Microsoft Quick Basic. Also for assembly language programmers and those wishing to interface to other languages, the fully commented source code, PDMA32.ASM, of the driver is provided. The driver is modularized and for special applications can be readily modified and re-assembled, though MetraByte can only support inquiries concerning operation of the driver in its unmodified form. The driver listing also provides examples of setting up the 8237 D.M.A. controllers and 8259 interrupt controllers as well as installing a typical interrupt handler.

5.2 Loading the Machine Language CALL Routine PDMA32.BIN

In order to make use of the CALL routine "PDMA32.BIN", it must first be loaded into memory. You must avoid loading it over any part of memory that is being used by the main body of your program or DOS or other programs which use high memory such as "RAM disks". If you do collide with another program, your computer will usually hang up although sometimes the results can be more bizarre. Often you will need to turn the power off and restore it to re-boot the

machine, the usual Ctrl-Alt-Delete reset may fail to restore DOS. This may sound ominous, but apart from the frustration, no damage will ever result!

PDMA32.BIN uses about 2.4 Kbytes of memory and is best loaded outside BASIC's workspace. A typical loading sequence is as follows:-

```
xx100 DEF SEG = &H4000          'segment of memory to load link
                                (choose an empty area e.g. @ 256K)
xx110 BLOAD "PDMA32.BIN",0      'load driver
  . .
Continue program
```

The above initializing steps will be the same for any interpreted BASIC program. A more comprehensive example is provided on the disk in DMA.BAS. Note that the DEF SEG = &H4000 statement in line 100 specifies the load location for the PDMA32.BIN driver. All subsequent CALL's will occur to the last DEF SEG address, so if you add other DEF SEG's in your program, remember to precede your CALL's to the PDMA-32 with the same DEF SEG = &H4000 that you used to load the link (see CALL and DEF SEG in your BASIC reference manual).

Finding a place to load PDMA32.BIN is seldom much of a problem now that most PC's are equipped with at least 512K of memory. The following explanation provides some insight into the process of choosing a memory location for the driver and what to do if memory is in short supply.

DOS occupies the bottom of memory, the amount of memory required being dependent on the version (it grows as each new revision adds extra features!). The simplified memory map below shows what happens after booting up BASICA.

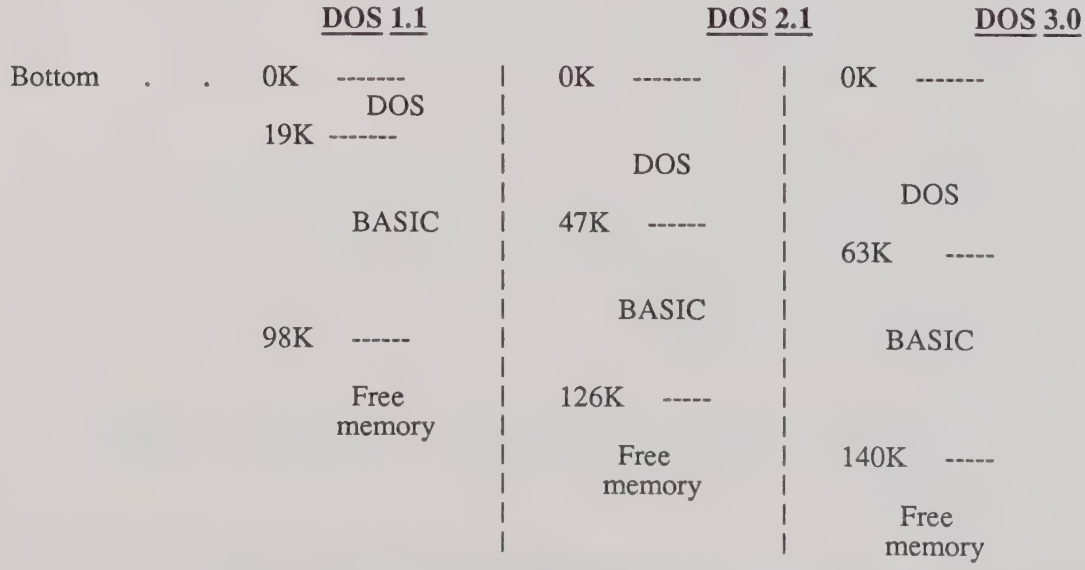


Fig. 5.1 Memory map with DOS and BASIC installed

PDMA32.BIN should be loaded somewhere in the free memory area so that it does not interfere with either BASIC or DOS. This would be above 98K (&H1880) for DOS 1.1, 126K (&H1F80)

for DOS 2.1 or 140K (&H2300) for DOS 3.0. If you have 512K (&H8000) or more of memory, then loading the link at DEF SEG = &H4000 or &H6000 is a good solution for all versions of DOS. One further small detail is that if you are using a PC compatible which does not have BASIC in ROM like the IBM, then BASIC is usually loaded as an .EXE file from the top of memory down. This is likely to fill up to 64K of the top segment of memory. Some virtual disks or print spoolers will do the same. Also if you are accustomed to using DOS resident programs such as Borland's Sidekick etc. be aware that these will push the loading floor of BASIC up and require a compensating increase in the location of PDMA32.BIN.

If you are memory limited, or you have so much resident stuff that there is no longer 64K left for BASIC to load in, then BASIC will attempt to make the most of what it can find. Instead of getting the message (typical example) when BASIC has loaded:-

```
The IBM Personal Computer Basic
Version A2.0 Copyright IBM Corp. 1981, 1982, 1983
60865 Bytes free
Ok
```

You may only get 49345 bytes free (or something less than 60000 bytes) for example. In this case make a note of what space BASIC has found. You can then contract this space further using the CLEAR function and load the link at the end of BASIC. This is more complicated, but just as effective.

Let's suppose we get the message 52000 bytes free. PDMA32.BIN will use 2.4 Kbytes, so to be on the safe side let's force BASIC to use 48K. Our initializing code would now be:-

```
xx100 CLEAR, 48000          'contracts BASIC workspace
```

Next we need to find out where BASIC has loaded in memory, add 48000 to it and load PDMA32.BIN just after the end of BASIC workspace. Absolute memory locations &H510 and &H511 always contain BASIC's load segment (true for IBM BASIC/BASICA & GWBASIC and other Microsoft derived BASICs):-

```
xx110 DEF SEG = 0                              'set up to read &H510 and &H511
xx120 LS = 256*PEEK(&H511)+PEEK(&H510)         'load segment
xx130 SG = LS + 48000/16                       'remember segment addresses are on
                                                16 byte (paragraph) boundaries
xx140 DEF SEG = SG                             'set up to load link
xx150 BLOAD "PDMA32.BIN",0                     'load link
  . .
Proceed with your program as before
```

5.3 Structure of the CALL Statement

If you are new to using CALL statements, this explanation may assist you in understanding how the CALL transfers execution to the machine language (binary) driver routine (also see CALL in your Basic Reference Manual). Prior to entering the CALL, the DEF SEG = SG statement sets the segment address at which the CALL subroutine is located. The CALL statement for the PDMA32.BIN driver must be of the form:-

xxxxx CALL PDMA32 (MD%, D%(0), FLAG%)

Let us examine the parameters after CALL one by one:-

PDMA32 - In interpreted BASIC this is a variable that specifies the offset of the start of our routine from the segment defined in the last DEF SEG statement. In our particular case, to keep matters simple, its value is always set to zero (PDMA32 = 0). In compiled BASIC (and most other compiled languages) PDMA32 has a different significance - it is the public name of the external routine that the linker will look for.

MD% - This is an integer variable that specifies the operation that we wish the driver to perform e.g. MD% = 0 initializes the driver, MD% = 1 sets up a port for a D.M.A. transfer etc. In the case of this driver, valid mode numbers range from 0 to 8.

D%(6) - This is a 7 element integer array that passes data to and from the driver. The significance of particular data items varies according to the mode (MD%) selected. Note that not all elements of D%(*) may be used in all modes.

FLAG% - Returns an error code if any of the specifying D%(*) or MD% are out of range or in the case of certain other error conditions. If there is no error, FLAG% is returned with a zero value.

The three variables within brackets are known as the CALL parameters. On executing the CALL, the addresses of the variables (pointers) are passed in the sequence written to BASIC's stack. The CALL routine unloads these pointers from the stack and uses them to locate the variables in BASIC's data space so data can be exchanged with them. Three important format requirements must be met:-

1. - The CALL parameters are positional. The subroutine knows nothing about the names of the variables, just their locations from the order of their pointers on the stack which in turn is controlled by the order that you write them within the parentheses.

   If you write:-

   xxxxx CALL PDMA32 (D%(0), FLAG%, MD%)

   you will mix up the CALL routine, since it will interpret D%(0) as the mode data, and MD% as the error variable etc. **The parameters must always be written in the correct order:-**

   (mode #, data, errors)

2. - The CALL routine expects its parameters to be of correct type and will write and read to the variables on this assumption:-

(integer, integer array, integer)

If you slip up and use the wrong variable types in the CALL parameters, the routine will not function correctly and may hang up the program.

3. - You **cannot** perform any arithmetic functions within the parameter list of the CALL statement. There can only be a list of variables. Also you are not allowed to replace variables by constants e.g. the following is ILLEGAL:-

```
zzzzz CALL PDMA32 (MD% + 3, D%(0), 6)
        (will generate a syntax error)
```

Apart from these restrictions, you can name the variables what you want, the names in the examples are just convenient mnemonics. Strictly, you should declare the variables before executing the CALL. If you do not, the simple variables will be declared by default on execution, but array variable obviously cannot be dimensioned by default and must be dimensioned before the CALL to pass data correctly when used as a CALL parameter. In the case of the integer array, the first element D%(0) should be specified as the data variable so that the CALL routine can locate all other data items in the array correctly.

5.4 Error Codes

Some value checking is performed on entry data and any errors discovered are returned in FLAG%. This is primarily to prevent you setting up the CALL with obviously incorrect data such as interrupt level 29, base address 2000, byte output data 299 etc. and is intended to help avoid a bad setup of the hardware which could hang the computer. If a non-zero error code is returned in any mode, execution of that mode will have been abandoned without action since error checking precedes any I/O to the hardware.

Error code #		Problem
0	:	No error, O.K.
1	:	Driver not initialized
2	:	Mode number <0 or >8
3	:	Invalid base address, <256 or >1008
4	:	Hardware error
5	:	D.M.A. level illegal, <0, 4 or >7
6	:	D.M.A. page wraparound error
7	:	D.M.A. data direction not 0 or 1
8	:	Interrupt level illegal, <3, 8, 13 or >15
9	:	Interrupt source out of range, <0 or >2
10	:	Interrupt slope not 0 or 1
11	:	Auto-recycle not 0 or 1
12	:	Byte or word specifier not 0 or 1
13	:	Digital output data out of range for byte I/O, <0 or >255
14	:	Port configuration code out of range, <0 or >1
15	:	Configuration change data not 0 or 1
16	:	Auxiliary output data out of range
17	:	Transfer clock source data illegal, not 0 or 1

Checking for errors is easily performed after each CALL and is recommended if not as a permanent feature then at least while debugging your program:-

```
xxx00 CALL PDMA32 (MD%, D%(0), FLAG%)
xxx10 IF FLAG%<>0 THEN PRINT "Error number ";FLAG%:STOP
```

5.5 CALL Modes

The PDMA32.BIN driver supports nine different modes (numbered 0 thru 8). Each mode performs a specific operation as described below:-

MD%	=	0	:	Initialize driver & test hardware
	=	1	:	Setup & perform D.M.A. transfer
	=	2	:	Return status of D.M.A. transfer
	=	3	:	Set timer rate
	=	4	:	Digital output
	=	5	:	Digital input
	=	6	:	Auxiliary output
	=	7	:	Setup & enable interrupt
	=	8	:	Disable interrupt

The method of use of each mode and the significance of D%(*) is described fully in the following subsections.

5.6 Mode 0: Initialization

Mode 0 initializes the driver and PDMA-32 hardware **and must be executed before using any of the other modes.** Initializing checks and stores the PDMA-32's base I/O address, the desired interrupt level for interrupts and the desired D.M.A. level. It also sets whether the PDMA-32's A and B ports will appear as a 16 bit or 8 bit peripheral for normal I/O.

The base I/O address is checked to be in the legal range of 256 - 1008 (Hex 100 - 3F0) for the PC/AT. If not an error exit #3 will occur. If O.K., the base I/O address is stored for use by other modes on re-entry to the CALL. A short read/write test is made to some of the PDMA-32 internal registers which is sufficient to detect the presence or absence of the board at the specified I/O address. If no board is detected (absent board, wrong base I/O address), error #4, hardware error is returned.

If initialization is successful, any other mode may be selected on subsequent CALL's. **Trying to select any other mode before performing initializing mode 0 will give rise to error # 1.** If an invalid D.M.A. level number or interrupt level is selected, errors #5 or #6 are returned.

Entry data:-
D%(0) - Base I/O address of PDMA-32, range 100-3F0H
D%(1) - Selected D.M.A. level, 0 - 3 and 5 - 7 are valid
D%(2) - Selected interrupt level, 3 - 7, 9 - 12 & 14 - 15 are valid
D%(3) - Selects port mode, 0 = byte I/O, 1 = word I/O
D%(4) thru D%(6) - Not used, value irrelevant

Exit data:-
D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8
3 if base address <100H or >3F0H
4 if hardware error, no board, incorrect I/O address
5 if D.M.A. level illegal
8 if interrupt level illegal
12 if byte/word mode specifier D%(3) not 0 or 1

A typical start of program initializing sequence would be:-

```
100 DEF SEG = &H4000                          'segment to load driver
110 BLOAD "PDMA32.BIN",0                      'load it at zero offset
120 PDMA = 0                                  'declare variables
130 MD% = 0                                   'select initialization
140 DIM D%(6)                                 'declare data array
150 FLAG% = 0
160 D%(0) = &H300                             'base address = 300 hex.
170 D%(1) = 5                                 'D.M.A. level 5
180 D%(2) = 9                                 'interrupt level 9
190 D%(3) = 1                                 'word mode - 16 bit I/O
200 CALL PDMA32 (MD%, D%(0), FLAG%)           'initialize
210 IF FLAG%<>0 THEN PRINT "Error in initializing # ";FLAG%:STOP
220 'continue program
```

5.7 Mode 1: Set up and Perform D.M.A. Transfer

Mode 1 sets up both the 8237 D.M.A. controller and the PDMA-32 for an input or output D.M.A. transfer using bytes or words. If bytes are transferred, Port A is used by the D.M.A. hardware, Port B remains open for use for programmed I/O and mode 1 will not disturb its configuration. In word mode, both ports A & B are used for D.M.A., port A transferring the less significant byte and port B the more significant.

The D.M.A. setup which the driver performs involves a number of steps:-

1. Disable any D.M.A. transfer already in progress.

2. Check data for valid ranges.

3. Check for a page wraparound problem. This can occur if for example you wanted to transfer 40,000 bytes starting at an offset of 32,000 in a page. The 8 bit D.M.A. page register selects 1 of 16 pages of 64 Kbytes out of the memory. The PC/AT system board hardware is not capable of incrementing the page register from the 8237 D.M.A. controllers when the end of a page is reached. Instead D.M.A. will continue at the beginning of the same page and may overwrite program or data memory. This condition of page wraparound is checked before enabling the D.M.A. operation.

4. The next step is to work out the correct control byte for the PDMA-32 D.M.A. control register.

5. The mode, initial address and byte count registers of the appropriate D.M.A. level of the 8237's are then loaded.

6. The timer is halted prior to enabling the D.M.A. channel so that the first transfer will always occur a fixed delay (corresponding to the timer pulse rate) after setup.

7. The 8237's mask register is enabled opening the D.M.A. channel.

8. The timer is brought up to its correct speed.

Return to your program then takes place. The D.M.A. transfer may continue as the rest of the following program executes. If you wish to determine the progress of a transfer, use mode 2.

Entry data:-

D%(0) - Number of transfers in bytes or words
D%(1) - Value irrelevant
D%(2) - Direction, 0 = input 1 = output
D%(3) - Auto-recycle on/off, 0 = off 1 = on
D%(4) - Transfer clock source, 0 = external 1 = timer
D%(5) - Transfer segment for D.M.A. buffer
D%(6) - Transfer offset for D.M.A. buffer

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8
6 D.M.A. page wrap around error
7 D.M.A. direction D%(2) not 0 or 1
11 Auto-recycle D%(3) not 0 or 1
17 Transfer clock source D%(4) not 0 or 1

A typical program entry preceding mode 1 might read:-

```
210 MD% = 1                                    'select mode number
220 D%(0) = 1000                               'transfer 1000 words
230 D%(1) = 1                                  'word transfer mode
240 D%(2) = 1                                  'output data on ports A & B
250 D%(3) = 1                                  'auto-recycle on - continuous output
260 D%(4) = 1                                  'clock data out with timer
270 D%(5) = &H5000                             'D.M.A. buffer memory segment
280 D%(6) = 0                                  'D.M.A. buffer memory offset
290 CALL PDMA32 (MD%, D%(0), FLAG%)  'execute mode 1
300 IF FLAG%<>0 THEN PRINT "Error in mode 1, # ";FLAG%:STOP
310 'continue program
```

A few points of explanation on the preceding example:-

D%(0) always sets the number of transfers whether they are bytes or words as selected by D%(1). If D%(0) = 0 then a full page (65,536) transfer will be performed.

D%(2) sets the port(s) direction, input or output. If the board has been set up (mode 0 initialization) to operate on any of the D.M.A. levels 0 - 3 (byte levels) only port A is used, otherwise word mode D.M.A. levels 5 - 7 use both ports A & B combined.

D%(3) controls whether or not the D.M.A. controller is set in the auto-initialize mode.

D%(4) controls selection of the D.M.A. transfer request pulse source. If D%(4) = 0 then the external XFER. REQ. on pin 2 of the rear connector is used. If D%(4) = 1 then transfer requests are generated by the internal timer. The timer rate can be set using mode 3.

D%(5) controls selection of the D.M.A. buffer segment in memory. If D%(5) = &HFFFF, then BASIC's data segment will be used (see caution below).

D%(6) controls the offset of the buffer area within the data segment set by D%(5).

D.M.A. DIRECTLY TO/FROM A BASIC INTEGER ARRAY

If you wish to transfer data directly to/from an integer array then proceed as follows. Performing D.M.A. directly to/from an array is a potentially dangerous procedure, read warning at end:-

```
210 MD% = 1                                    'select mode number
220 N = 360                                    'number of elements in array
230 DIM ARRAY%(N)
240 D%(0) = N+1                                'transfer N+1 words
250 D%(1) = 1                                  'must be word transfer mode
260 D%(2) = 1                                  'output data on ports A & B
270 D%(3) = 1                                  'auto-recycle on
280 D%(4) = 1                                  'clock data out with timer
290 D%(5) = &HFFFF                             'array segment = BASIC's
300 D%(6) = VARPTR(ARRAY%(0))                  'array offset
310 CALL PDMA32 (MD%, D%(0), FLAG%)  'execute mode 1
320 IF FLAG%<>0 THEN PRINT "Error in mode 1, # ";FLAG%:STOP
330 'continue program
```

Warning !

Since BASIC dynamically allocates variable storage and locates arrays above simple variables, adding a new simple variable after line 310 will unavoidably relocate ARRAY%(*). The D.M.A. hardware will not be aware of this and will carry on outputting data from the locations it got in lines 290 & 300 leading to strange effects especially on input of data. You must use a structured programming method of pre-declaring all variables at the start of your program to safely perform D.M.A. directly in/out of BASIC arrays. If in doubt, use a buffer area external to BASIC.

5.8 Mode 2: Return Status

Mode 2 returns information about the status of D.M.A. and interrupt operations. For D.M.A., D%(3) indicates whether a D.M.A. transfer set up by mode 1 is still active. It also returns the number of transfers (bytes or words) requested in mode 1 in D%(1) and the number transferred so far in D%(0). If the D.M.A. controller is operating in non-auto initialize mode and the transfer is complete (number transferred = number requested), then D%(3) returns 0 to indicate D.M.A. inactive. It is also possible for the hardware to generate a terminal interrupt to indicate the end of a D.M.A. transfer, although this mode does not utilize this capability.

Mode 2 also returns the current data directions of ports A and B and the current interrupt status whether active or done.

Entry data:-

D%(0) thru D%(6) - value irrelevant

Exit data:-

D%(0) - Number of bytes/words transferred
D%(1) - Number of byte/word transfer requested
D%(2) - Auto-initialize on/off, 1 = on 0 = off
D%(3) - D.M.A. active/finished, 1 = active 0 = finished
D%(4) - Port A data direction, 0 = input 1 = output
D%(5) - Port B data direction, 0 = input 1 = output
D%(6) - Interrupt status, 0 = disabled 1 = enabled
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8

A typical programming sequence using mode 2:-

```
310 MD% = 2                                    'select mode number
320 CALL PDMA32 (MD%, D%(0), FLAG%)   'execute mode 2
330 IF FLAG%<>0 THEN PRINT "Error in mode 2, # ";FLAG%:STOP
340 IF D%(3) = 1 THEN GOTO 320                 'wait till D.M.A. finished
```

5.9 Mode 3: Set Timer

Mode 3 sets the division ratio for the timer. The timer output consists of two 16 bit down counters driven in cascade from a 10.000MHz 0.01% precision xtal clock. Initializing (mode 0) sets these counters to operate in 8254 mode 2 (see Appendix B) or rate generator mode which produces a single pulse on terminal count of counter 1. Mode 0 also initially loads these counters to 65,535 (FFFFH) so that the initialized output rate is about 1 pulse every 7 minutes. Immediately after initialization the counter output is guaranteed to be low for 7 minutes which allows plenty of time for set up of an external device.

This mode re-loads the counters to any other desired rate up to a maximum of 2.5MHz. The output pulse rate is:-

Rate = (10,000,000)/(D%(0) * D%(1)) pulses/sec.

Permissible values for D%(0) and/or D%(1) are 2 thru 65,535. Values of 0 and 1 are not legal.

It is also possible to operate the counters in square wave mode (8254 mode 3) but this requires direct programming to the 8254 control port as this driver does not support changing the initialized configuration. Counting may also be enabled and disabled with the TIMER GATE input (pin 3). When held low, this input inhibits counting, when open or held high counting is allowed to proceed.

Entry data:-

D%(0) - Counter 0 divider
D%(1) - Counter 1 divider
D%(2) thru D%(6) - not used

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8

A typical timer setup to output 1 KHz would be:-

```
340 MD% = 3                                          'select mode number
350 D%(0) = 1000                                     'counter 0 divisor
360 D%(1) = 10                                       'counter 1 divisor
360 CALL PDMA32 (MD%, D%(0), FLAG%)   'execute mode 3
370 IF FLAG%<>0 THEN PRINT "Error in mode 3, # ";FLAG%:STOP
380 'continue program
```

Note:-

The output pulse width from counter 1 depends on the period of the clock it receives from counter 0. In this example, counter 0 outputs a frequency of 10,000/1000 = 10KHz so the output pulse width will be 100 microseconds. Since counter 1 divides by 10 the ultimate timer rate is 1KHz. If D%(0) = 100 and D%(1) = 100, the timer rate would still be 1KHz but the pulse width would be 10 microseconds.

5.10 Mode 4: Digital Output

Mode 4 performs digital output as a programmed transfer either through the A port or the B port or in the case of word transfers, through both the A & B ports. Note this mode does not perform D.M.A., use mode 1 instead. The operation performed by this mode is equivalent to a direct write to the port address using OUT addr, data in most languages. Once written, data is latched on the output port and can be read back using mode 5 or INP(addr). The operating mode of the ports (word/byte) is set by variable D%(3) in initializing mode 0. If word mode has been selected, then the PDMA-32 appears as a 16 bit peripheral for digital I/O and data is written in a single instruction from ports A & B combined. If byte I/O mode has been selected then ports A & B are written separately as byte wide ports and the PDMA-32 appears as a byte oriented peripheral to the PC/AT bus. Note it is possible to be doing D.M.A. in word mode and still perform byte wide I/O operations on the A & B ports.

The output port(s) are selected by D%(1). Byte (8 bit) transfers may be made either through the A port or the B port. If the PDMA-32 is operating in word I/O mode, then the value of D%(1) is irrelevant. Word (16 bit) transfers are made through both the A and B ports with the A port outputting the least significant byte and the B port outputting the most significant.

The data direction of the ports is checked initially by reading the PDMA-32's D.M.A. control register. If it differs from the desired output arrangement, the data directions are set accordingly. On power up, all ports are reset to the input condition.

A check on valid data is performed on byte transfers only and error code #13 is returned if out of range and the output operation abandoned.

Entry data:-

D%(0) - output data, range 0 to 255 if byte
-32768 to 32767 if word
D%(1) - output port(s), value irrelevant in word I/O mode
0 = port A (byte)
1 = port B (byte)
D%(2) thru D%(6) - not used

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8
13 if data D%(0) is out of range, <0 or >255 for byte xfers.
14 if port configuration D%(1) data <0 or >1

The following example outputs all 1's (65,535) as a word on ports A & B. Note that we wish to output unsigned data, whereas integers store signed data - see Appendix A for further explanation. The examples assume we have performed initialization , mode 0, with D%(3) = 1 (word mode). Initially we start with the data in a real variable D and sign correct it for output in an integer in line 400:-

```
380 MD% = 4                                    'select mode number
390 D = 65535                                  'data to output
400 IF D > 32767 THEN D = D - 65536            'sign correct
410 D%(0) = D                                  'output data
420 CALL PDMA32 (MD%, D%(0), FLAG%)  'execute mode 4
430 IF FLAG%<>0 THEN PRINT "Error in mode 4, # ";FLAG%:STOP
440 'continue program
```

The following code would output a travelling pattern of 1 on port B assuming byte I/O mode had been selected in initializing mode 0 (D%(3) = 0):-

```
380 MD% = 4                                    'select mode number
390 D%(1) = 1                                  'select port B
400 FOR I% = 0 TO 7
410 D%(0) = 2^I%                               'output data
430 CALL PDMA32 (MD%, D%(0), FLAG%)  'execute mode 4
440 IF FLAG%<>0 THEN PRINT "Error in mode 4, # ";FLAG%:STOP
450 NEXT I%
460 'continue program
```

5.11 Mode 5: Digital Input

Mode 5 reads byte (8 bit) data from port A or port B as specified by D%(1), or word (16 bit) data from ports A and B. In the case of word data reads, port A provides the least significant byte, and port B the most significant. Data is always returned in D%(0). The operating mode of the ports (word/byte) is set by variable D%(3) in initializing mode 0. If word mode has been selected, then the PDMA-32 appears as a 16 bit peripheral for digital I/O and data is read in a single instruction from ports A & B combined. If byte I/O mode has been selected then ports A & B are read separately as byte wide ports and the PDMA-32 appears as a byte oriented peripheral to the PC/AT bus. Note it is possible to be doing D.M.A. in word mode and still perform byte wide I/O operations on the A & B ports. If word mode has been selected on initialization, the value of D%(1) is irrelevant.

Note that the ports may be in input or output mode on entry to mode 5. If D%(2) is 0, mode 5 will not interfere with the port configuration i.e. will read back data from a port set in output mode by mode 4 and leave it set in output mode. If however you want to alter the configuration and make the port an input port only, D%(2) should be set to 1. An addressed port will then the be set in input mode prior to reading it and will remain in input mode afterwards until changed by an output operation using mode 4. Ports are automatically configured as inputs by the computer on power up reset.

Entry data:-

D%(0) - value irrelevant
D%(1) - input port(s), value irrelevant if in word I/O mode
0 = port A (byte)
1 = port B (byte)
D%(2) - change configuration
0 = no change
1 = change addressed port to input mode
D%(3) thru D%(6) - not used

Exit data:-

D%(0) - data from port
D%(1) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8
14 if port configuration (D1) data <0 or >1
15 if configuration change data (D2) not 0 or 1

Typical programming example, set port A to input and read (assumes PDMA-32 initialized in byte I/O mode):-

```
460 MD% = 5                                    'select mode number
470 D%(1) = 0                                  'select port A
480 D%(2) = 1                                  'change port A to input config.
490 CALL PDMA32 (MD%, D%(0), FLAG%)  'execute mode 5
500 IF FLAG%<>0 THEN PRINT "Error in mode 5, # ";FLAG%:STOP
510 PRINT "Data from port A = ";D%(0)
520 'continue program
```

Second example, set port B to output, output data and read it back:-

```
380 MD% = 4                                          'select digital output mode
390 D%(0) = 99 : X% = D%(0)                          'output data and save in X%
400 D%(1) = 1                                        'set port B to output byte
410 CALL PDMA32 (MD%, D%(0), FLAG%)                  'execute mode 4
420 IF FLAG%<>0 THEN PRINT "Error in mode 4, # ";FLAG%:STOP
430 MD% = 5                                          'select digital input mode
440 D%(1) = 1                                        'select port B
450 D%(2) = 0                                        'don't change port B config.
460 CALL PDMA32 (MD%, D%(0), FLAG%)                  'execute mode 5
470 IF FLAG%<>0 THEN PRINT "Error in mode 5, # ";FLAG%:STOP
480 IF X% <> D%(0) THEN PRINT "Error on port B"
490 'continue program
```

5.12 Mode 6: Auxiliary Output

Mode 6 provides a means of writing data to any of the auxiliary data bits of the D.M.A. or interrupt control registers. Data is OR'ed into the registers so that other control bits are not altered.

If any of the data in D%(0) thru D%(2) is not 0 or 1, error code 16 is returned.

Entry data:-

D%(0) - AUX 1 data (0 or 1 permitted)
D%(1) - AUX 2 data (0 or 1 permitted)
D%(2) - AUX 3 data (0 or 1 permitted)
D%(3) thru D%(6) - not used

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8
16 if any auxiliary data bit is not 0 or 1

Example of typical use:-

```
520 MD% = 6                                    'select mode number
530 D%(0) = 0                                  'AUX 0 = 0
540 D%(1) = 1                                  'AUX 1 = 1
550 D%(2) = 1                                  'AUX 2 = 1
560 CALL PDMA32 (MD%, D%(0), FLAG%)   'execute mode 6
570 IF FLAG%<>0 THEN PRINT "Error in mode 6, # ";FLAG%:STOP
580 'continue program
```

Note: There is no mode provided in the driver for reading back the state of the AUXiliary control bits although since both the D.M.A. and interrupt control registers are R/W registers, it is possible to do this. The following little program will perform this function:-

```
590 BASE = &H300                              'provide Base I/O address
600 IF (INP(BASE + 2) AND &H10)=&H10 THEN AUX1=1 ELSE AUX1=0
610 IF (INP(BASE + 2) AND &H20)=&H20 THEN AUX2=1 ELSE AUX2=0
600 IF (INP(BASE + 3) AND &H8)=&H8 THEN AUX3=1 ELSE AUX3=0
```

5.13 Mode 7: Interrupt Enable

Mode 7 installs an interrupt handling routine and enables a hardware interrupt on the level previously selected in initialization mode 0. An interrupt may be generated from 1 of 3 sources, an external interrupt on rear connector pin 1, the internal PDMA-32 timer, or a terminal interrupt from the selected D.M.A. level. The source is selected by D%(0). In the case of external interrupts, an interrupt will be generated on the positive edge of the signal if D%(0) = 0 or the negative edge if D%(0) = 1. After the edge initiated interrupt, it is a requirement of the 8259 interrupt controllers in the PC/AT that the interrupt remain asserted until the interrupt service routine starts execution. The PDMA-32 hardware includes an internal interrupt latch which should be cleared at some point in your interrupt handler by reading the interrupt status register at I/O address BASE + 0AH.

The various steps performed by mode 7 are:-

1. Check for valid source data
2. Disable any active interrupt on the selected level
3. Install interrupt vectors using DOS function call 25H of INT 21H
4. Write correct control byte to PDMA-32 interrupt control register
5. Enable the 8259 interrupt controllers by clearing the appropriate mask register bit for the level selected. The interrupt level is selected during initialization mode 0 and is stored in the interrupt level select register.

A sample "beep the bell" interrupt service routine is supplied in the PDMA32.ASM source listing (labelled INTH:) and assembled into the PDMA32.BIN driver. You can modify this interrupt service routine to do whatever you wish. To develop your own routine, use any text editor to expand and modify the code and re-assemble with the Macro Assembler following the instructions in the file HOWTOBIN.DOC to generate a BASIC callable .BIN file.

The PDMA-32 can generate a hardware interrupt on any of the 11 expansion bus levels 3 - 7, 9 - 12 & 14 - 15. The level is software selectable through the interrupt level select register which is set during mode 0 initialization. The PC/AT's 8259 interrupt controllers can prioritize 15 different hardware interrupts. Level 0 is the highest priority and is used by the internal timer which generates an interrupt about 18 times/sec. This is used by the BIOS and DOS to provide the system time and date. Level 1 is used by the keyboard to signal that a key has been pressed and invoke a keyboard handling routine. Level 2 is used to cascade the second 8259 controller. Levels 0, 1, 2, 8 and 13 are internal to the PC/AT and not available on the expansion bus. The interrupt levels have been assigned by I.B.M. to the standard peripherals as follows:-

Level 0	:	Used for interval timer, time of day (not on bus)
Level 1	:	Used by keyboard (not on bus)
Level 2	:	Used for cascade input on PC/AT only (not on bus)
Level 3	:	Used by COM2: serial port if installed
Level 4	:	Used by COM1: serial port if installed
Level 5	:	Used by LPT2: or hard disk if installed
Level 6	:	Used by floppy disk drive adapter
Level 7	:	Used by LPT1:
Level 8	:	Used internally by real time clock (not on bus)
Level 9	:	Unassigned
Level 10	:	Unassigned
Level 11	:	Unassigned
Level 12	:	Unassigned
Level 13	:	80287 coprocessor error (not on bus)
Level 14	:	Fixed disk drive controller
Level 15	:	Unassigned

Generally the unassigned levels 9 - 12 & 15 are the best choices, but you can use any other level if you know that the corresponding peripheral device is not installed E.G. if you have no COM2: serial port, level 3 would be free. It is best to avoid multiplexing one interrupt level between 2 or more adapters although this can be done with the restriction that only one adapters interrupt can be active at a time. Do not use levels 6 & 14 which are usually in use by the disk drives.

Since the upper levels 8 - 15 are cascaded through interrupt level 2, they actually have a higher priority than levels 3 - 7. If no lower level interrupt is pending and interrupts are enabled, an interrupt will normally be serviced within a few microseconds of its generation. If a lower level interrupt collides with a simultaneous one from the PDMA-32, it can delay servicing of the PDMA-32 interrupt. The usual culprit here is the timer interrupt on level 0, it can occasionally delay the PDMA-32 interrupt by 30-40 microseconds, which in most cases is negligible but in some cases may be a problem. If it is a nuisance, the delay variation can be eliminated by disabling the timer interrupt through the 8259 mask register (located at I/O port &H21), and also by refraining from using the keyboard or COM: ports.

Once a PDMA-32 hardware interrupt has been enabled by mode 7 it will remain active until disabled by mode 8 or stopped by a "self extinguishing" interrupt service routine i.e. one that does the equivalent of mode 8 after a certain number of operations.

Entry data:-

D%(0) - Specifies interrupt source:-
- 0 = External input, positive slope
- 1 = External input, negative slope
- 2 = D.M.A. terminal interrupt
- 3 = Timer interrupt

D%(1) thru D%(6) - not used

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
- 1 if mode number <0 or >8
- 18 if interrupt source data (D0) out of range, <0 or >3

The following program will beep the bell ad nauseam every 2 seconds:-

```
520 MD% = 3                                          'set timer
530 D%(0) = 20000                                    'counter 0 divisor
540 D%(1) = 1000                                     'counter 1 divisor - 0.5Hz
550 CALL PDMA32 (MD%, D%(0), FLAG%)                  'execute mode 3
560 IF FLAG%<>0 THEN PRINT "Error in mode.3, # ";FLAG%:STOP
570 MD% = 7                                          'enable interrupt
580 D%(0) = 3                                        'from timer
590 CALL PDMA32 (MD%, D%(0), FLAG%)                  'execute mode 7
600 IF FLAG%<>0 THEN PRINT "Error in mode 7, # ";FLAG%:STOP
610 'continue program, try mode 8 to shut it off!
```

Note: Mode 2 provides information on whether the interrupt is active if there is any doubt.

5.14 Mode 8: Interrupt Disable

Mode 8 sets the appropriate bit in the 8259 interrupt controllers mask register to disable further interrupts for the level selected in mode 0. This driver does not restore previous interrupt routine vectors (possibly required if sharing an interrupt level with another device). Modes 7 & 8 can be expanded to do this if required.

Entry data:-

D%(0) thru D%(6) - value irrelevant

Exit data:-

D%(0) thru D%(6) - Unchanged
FLAG% - 0 if o.k., otherwise:-
1 if mode number <0 or >8

To turn off the interrupt started in mode 7's example:-

```
610 MD% = 8                                          'select mode
620 CALL PDMA32 (MD%, D%(0), FLAG%)      'execute mode 8
630 IF FLAG%<>0 THEN PRINT "Error in mode 8, # ";FLAG%:STOP
640 'continue program
```

5.15 Example BASIC Programs

Several example BASICA programs are provided on the disk. They are commented and can be listed to provide further information. DMA.BAS shows you how to load PDMA32.BIN, initialize the PDMA-32, and perform D.M.A. transfers with the PDMA-32.

The remaining programs have self explanatory file names:-

1. TIMER.BAS sets the timer to any rate
2. STATUS.BAS returns the status
3. IN.BAS reads ports
4. OUT.BAS writes data to ports
5. INT.BAS enables and disables interrupts.

Simply list these programs for further explanation.

5.16 Compiling a BASIC Program

After you have your program running in interpreted BASIC, you may find that you need to increase its execution speed. Interpreted BASIC requires several milliseconds to execute each statement, and a sizable program particularly with extensive analysis or data processing loops can take a good part of a second to execute. One of the simplest ways to speed things up is to compile your program using the IBM Basic Compiler, or Microsoft Basic Compiler or Quick Basic Compiler. In the case of digital I/O, this will usually lead to an improvement in execution speed by a factor of 100 to 1000 times depending on the type of operations in the program. Of course, it will not have any effect on D.M.A. transfer rates which are determined by the hardware, but D.M.A. setup will be faster.

The file PDMA32.OBJ is supplied on the software disk. This is the object code version of PDMA32.BIN and will be needed when running the linker.

Before you charge into compiling your program, spend some time familiarizing yourself with the operation and capabilities of the Basic Compiler. Try out a few examples and get used to some of the differences from interpreted BASIC e.g. you must declare all arrays prior to use etc. When you are ready proceed as follows:-

1. Take your interpreted BASIC program and delete all lines that load PDMA32.BIN e.g. the DEF SEG and BLOAD.
2. Store your program in ASCII form (SAVE "yourprog.bas",A).
3. Compile it (BASCOM yourprog) with whatever compiler switches you require. Fix any errors before proceeding.

4. Next run the linker (it is best to use the one supplied with your compiler rather than DOS).

```
A> LINK yourprog.obj + PDMA32.OBJ

IBM Personal Computer Linker
Version 1.10 (C)Copyright IBM Corp. 1982

Run File [MYPROG.EXE]
List File [NULL.MAP ]
Libraries [.LIB]
```

At the end of the linking session, you will have a DOS executable file MYPROG.EXE.
If you receive an "Unresolved External" error message from the linker, it is most likely that you have either omitted or misnamed the PDMA32.OBJ declaration when prompted for the files to link, or that the calls do not read CALL PDMA32 (you must use the name PDMA32 in the call). In compiled BASIC, PDMA32 now is the name of an external routine and not an offset variable as in interpreted BASIC.

5.17 Multiple PDMA-32's in One System

What if you wish to operate more than one PDMA-32 in a system? To avoid conflicts, each PDMA-32 must have a different base I/O address and for simultaneous operation be set to different interrupt and D.M.A. levels. It is also possible to share D.M.A. and interrupt levels if the operations can be sequential instead of simultaneous. Each board must also be assigned its own CALL routine. To do this start by loading the PDMA32.BIN routine at different locations in memory:-

```
xxx10 SG1 = &H3000
xxx20 SG2 = &H4000
xxx30 DEF SEG = SG1
xxx40 BLOAD "PDMA32.BIN",0
xxx50 DEF SEG = SG2
xxx60 BLOAD "PDMA32.BIN",0
```

Now the CALL appropriate to each board can be entered as required. Note that each CALL is preceded by a DEF SEG appropriate to that board:-

```
yyy10 DEF SEG = SG1
yyy20 CALL PDMA32 (MD1%, D1%(0), FLAG1%)
yyy30 DEF SEG = SG2
yyy40 CALL PDMA32 (MD2%, D2%(0), FLAG2%)  'etc.
```

5.18 Assembly Language Programming

The source code, PDMA32.ASM, for the driver is also supplied on disk. This can be printed out (use the Ctrl-PrtSc and TYPE PDMA32.ASM) or examined and altered with any text editor e.g. Wordstar. It is heavily commented and self explanatory.

If you wish to link this with another assembly language program or another language with different call conventions e.g. C, Pascal, Fortran etc., you will need to modify the entry and exit modules which are designed for the CALL procedure used with BASIC. The routines for each mode are self-contained and communicate only with the local variables internal to the driver and defined in the source, making it easy to lift them out unchanged.

Interfacing to calls from C, Pascal and Fortran involves altering the entry and exit routines as these languages usually pass pointers consisting of segment:offset addresses on the stack (unlike BASIC which passes offsets only).

The steps involved in creating a BASIC loadable .BIN driver are described in HOWTO.DOC. Basic tacks a header onto a BLOADable file, the method of using the linker, EXE2BIN, debug and the Basic interpreter achieve this are detailed in this file. Use TYPE HOWTO.DOC, using Ctrl-NumLock to stop/start scrolling or list on the printer with Ctrl-PrtSc.

Section 6

APPLICATIONS INFORMATION

6.1 Typical Handshake Connection

This section describes the method of connection and operation of the PDMA-32 and then expands to some more specific examples using A/D and D/A converters.

Whether you are transferring bytes or words into or out of the PDMA-32, the transfer control signals, XFER. REQ. and XFER. ACK. work the same way. Their timing is shown below:-

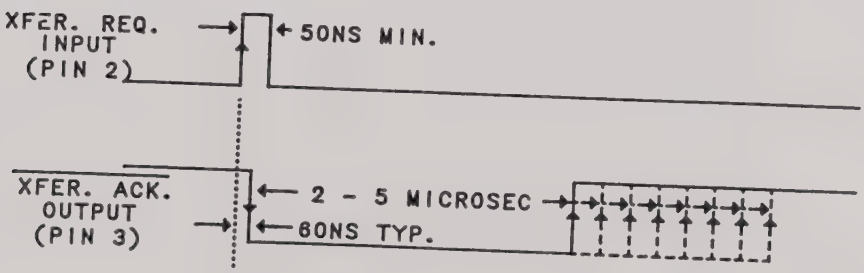


Fig. 6.1 XFER. REQ. / XFER. ACK. Timing

For inputs to the PDMA-32, the data should be presented to the port(s) and when stable, the XFER.REQ. input should be taken high. About 60nS later, the XFER. ACK. will go low indicating that the PDMA-32 has initiated a D.M.A. request cycle. The input data must remain stable until XFER. ACK. returns high indicating completion of the transfer cycle. The next byte or word of data can then be presented to the PDMA-32 and the cycle repeated to store it in the next memory location. This handshake is the same whether you are transferring bytes or words. The XFER. ACK. will typically go low for at least 2.4 microseconds on PC/AT's with 8237 D.M.A. controllers clocked at 3MHz and may stay low for up to 5 microseconds on some transfers due to D.M.A. latency.

For outputs from the PDMA-32, when the next byte or word of data is required, the XFER REQ. input should be taken high. The XFER. ACK. will go low on receipt of the transfer request and remain low until the byte or word has been transferred and the new data is valid. The data then remains latched on the output ports until the next transfer request.

If you want to clock data out at a steady rate, the internal 8254 timer may be substituted for the external XFER. REQ. by appropriate programming of the D.M.A. control register. The timer can be set to a suitable rate. Note that the timer pulse that generates the transfer request does not appear on the XFER. REQ. input, this input is inactive, but the same timer pulse that generates the internal transfer request is always present on the TIMER OUT. This is sometimes useful as a reference signal or in double buffering the output data and re-clocking to eliminate jitter due to D.M.A. latency. Timer operation can be suspended by taking the TIMER GATE input low and will continue from where it left off on taking the TIMER GATE input high again.

The A DIR. and B. DIR. port direction outputs provide information on the signal direction of the ports and if required can be used to activate bi-directional drivers for re-buffering the output data. This can let you perform a series of D.M.A. outputs to one device connected to the ports followed by a series of D.M.A inputs from another on the same ports. The direction signals can be used to enable the right devices as appropriate. In word mode I/O, the direction of both A & B ports is provided by the A DIR. output (ignore B DIR.).

When enabled through the interrupt control register, the INTERRUPT INPUT can feed an external interrupt signal through to any of the PC/AT expansion bus interrupt levels. The external interrupt pulse may be as short as 100nS as it triggers a monostable in the PDMA-32 which sets a latch to insure correct interrupt service operation. The latch should be cleared during execution of the interrupt handler by reading the interrupt status register at I/O address BASE + 0AH. Interrupts can be generated on negative or positive edges of the external signal, the slope being selected through the interrupt control register. It is also possible to select the timer output or the D.M.A. controller terminal count pulse as interrupt sources through software control via the interrupt control register.

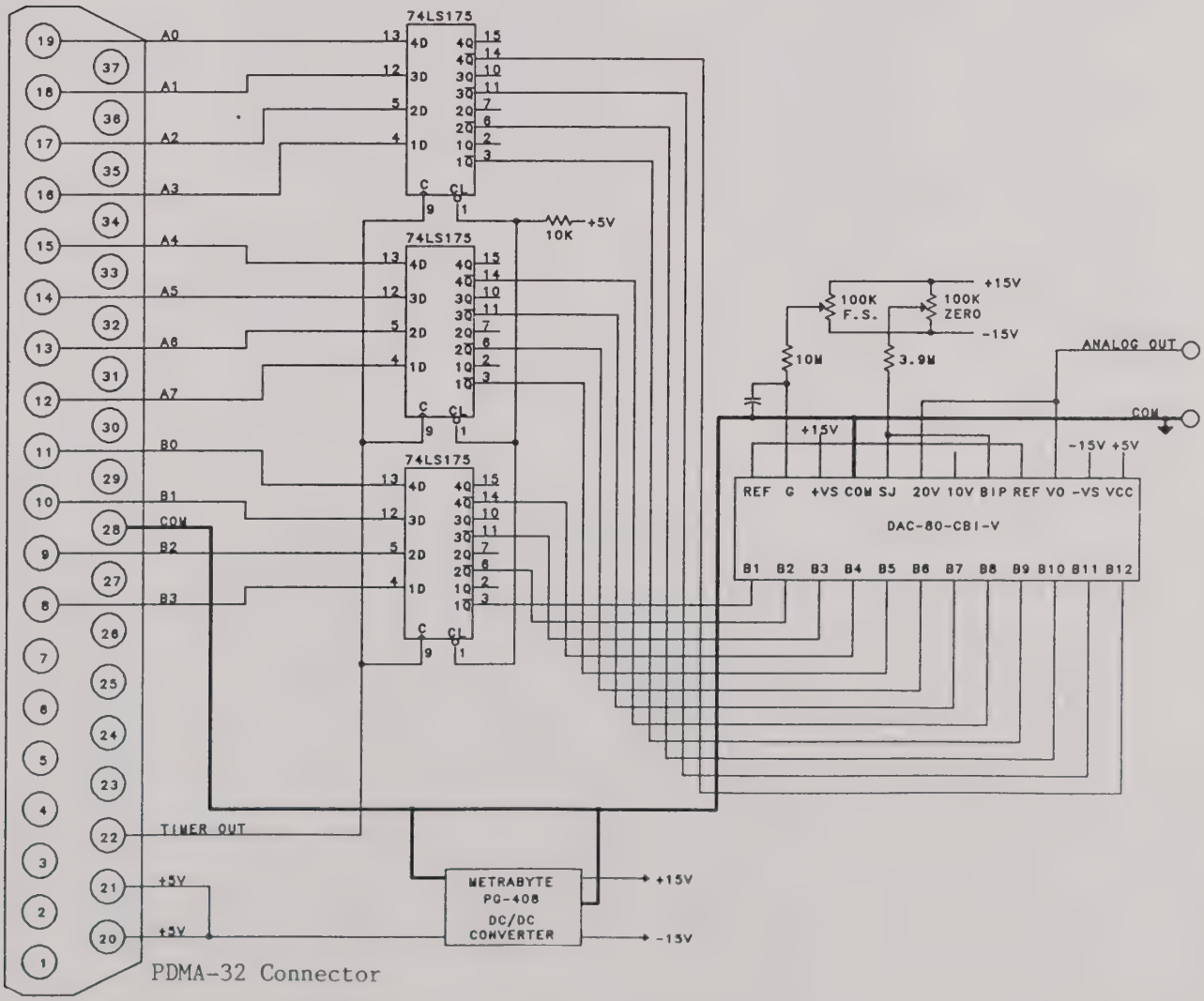
The auxiliary outputs (AUX1-3) provide three extra output bits that are user programmable and may be useful in selecting and controlling external devices. Since these bits are part of the D.M.A. and interrupt control registers, it is necessary to first read these registers and OR the auxiliary data in, so that none of the other register bits are affected. The PDMA32.BIN driver does this operation automatically.

Power from the computer +5v supply is available from the PDMA-32 connector. This can be used to power external buffering logic, small peripheral devices and DC/DC converters as well as providing a connection point for pull-up and terminator resistors if used. When using the internal computer power be careful not to apply external power sources such as line voltage to the connections as this could cause very considerable damage to your computer. Although the +5v supply is a convenient feature, it should be used with caution. Use an external power supply if there is any possibility of abuse. The +5v supply is well protected against short circuits which will result in a shut down of the whole computer. To restore power, switch off the computer, remove the short circuit and switch the computer back on again. The amount of power available is dependent on the other peripheral devices in the computer, but in any case it is recommended that you limit current drawn to less than 2 amps due to copper trace and connector limitations on the PDMA-32.

6.2 Waveform Generation with a D/A Converter

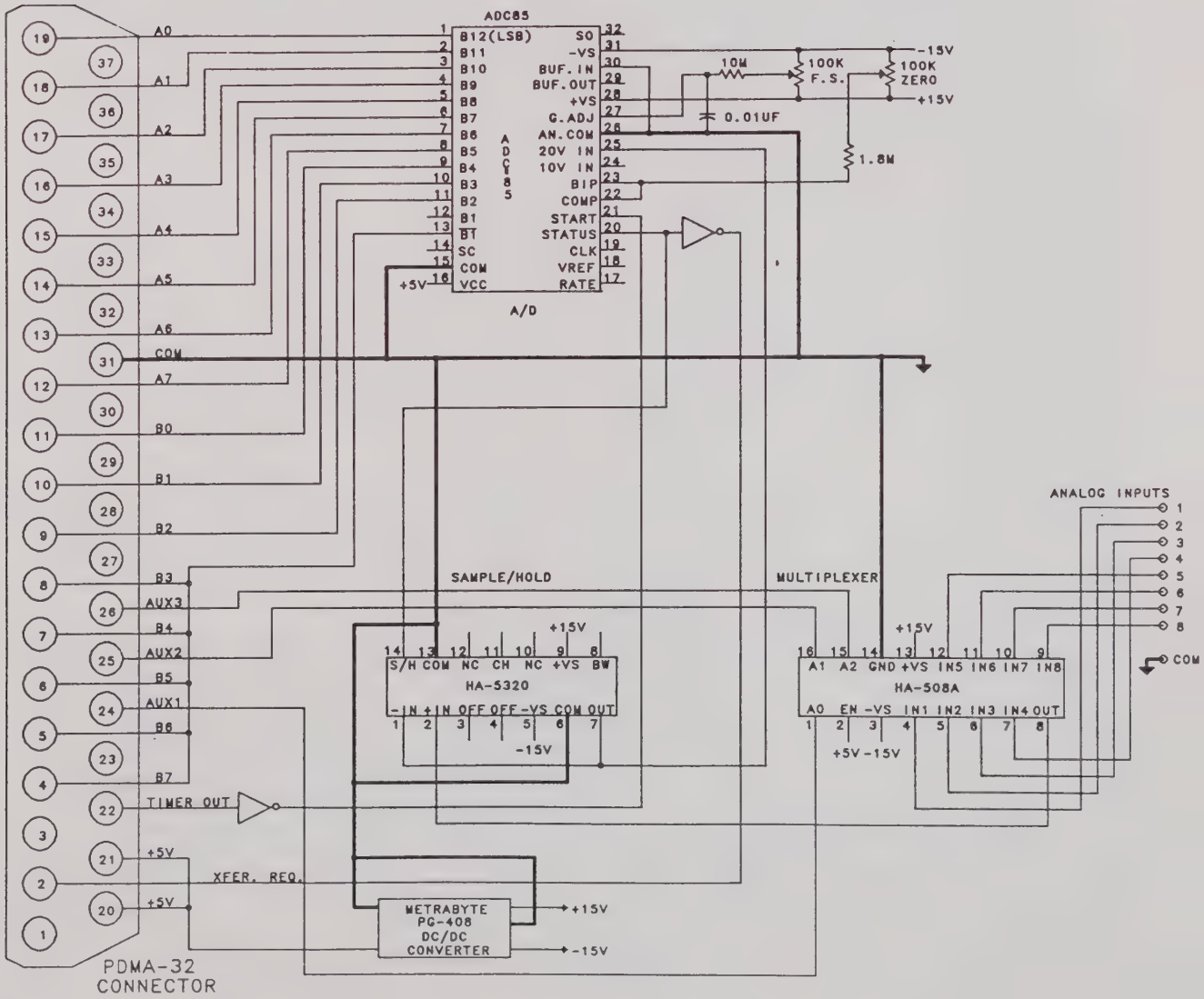
Note the use of an external latch to re-synchronize the data to the internal timer clock. This avoids any time jitter due to D.M.A. latency although the D/A output will always be one timer clock period behind the output of the PDMA-32. Data can be output single shot or continuously using Mode 1 of the PDMA32.BIN driver The schematic below is a working example, other components e.g. 16 bit D/A's can be substituted.

The PDMA-32 delivers +5v power from the computer on pins 20 & 21 of the connector. This can be used to power the logic circuits of the external interface and a small DC/DC converter such as MetraByte's PG-408 could also generate the +/-15v analog supplies for the D/A from the +5v power. Alternatively an external supply may be used.



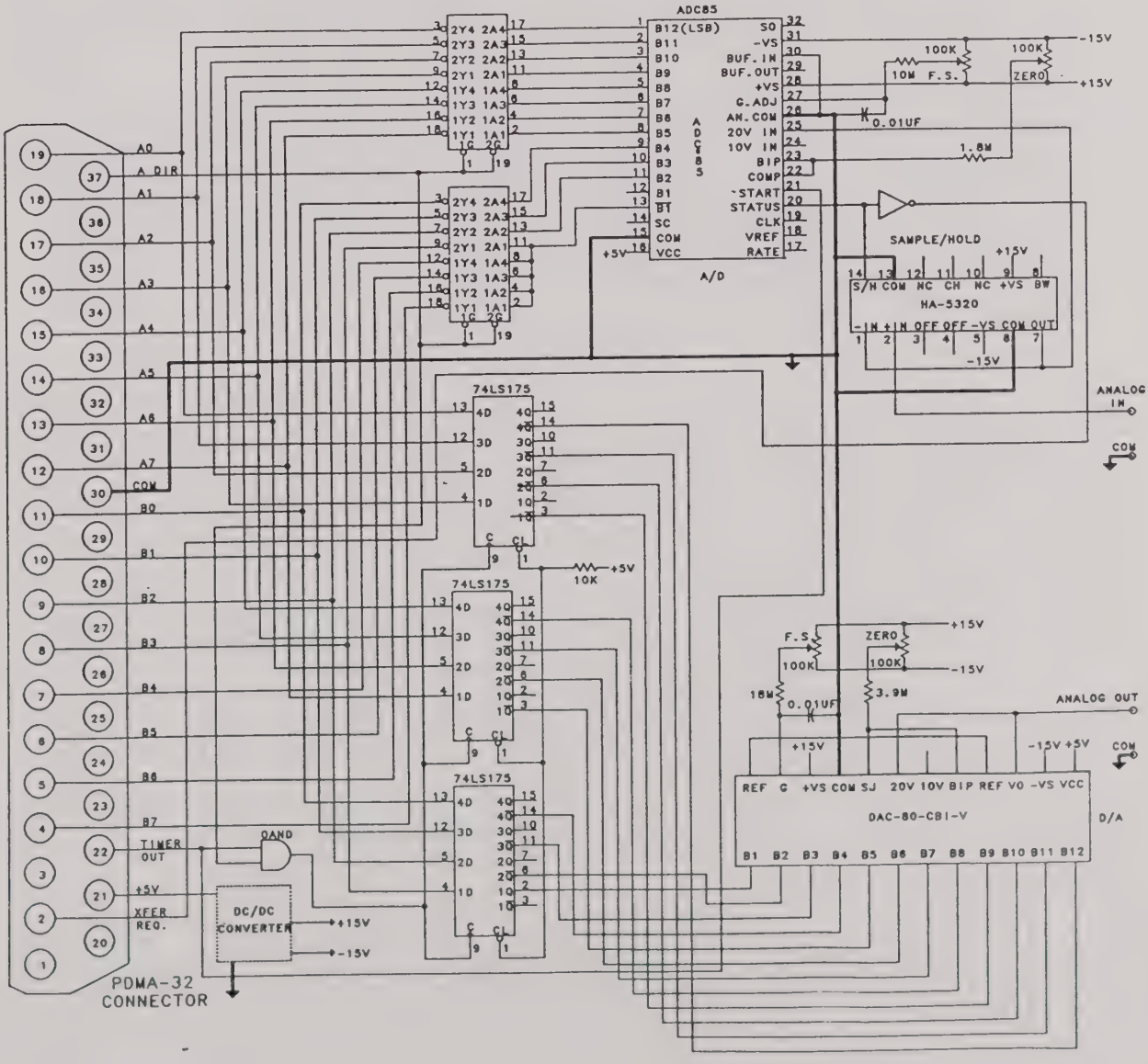
6.3 High Speed A/D Conversion

The schematic below shows connection of a 12 bit A/D converter (16 bit A/D's could also be used). The A/D is triggered by the PDMA-32 timer and the A/D's EOC (end of conversion) signal generates the DMA XFER. REQ. after each conversion. Rates in excess of 100KHz are readily obtained using Mode 1 of the PDMA32.BIN driver.The AUX outputs are used to select 1 of 8 channels using an input multiplexer.



6.4 Combined A/D and D/A Conversion using Directional Controls

This combines the previous 2 examples and uses the A DIR direction output signal to control whether the A/D or D/A converter is enabled. With this circuit it is possible to digitize an event at high speed with the A/D and reconstruct it back through the D/A either at the same speed or a reduced speed to drive a plotter for instance. A minimum of external circuitry is required.



6.5 Commonly Encountered Problems

This is a brief guide to some of the most common application problems that are most likely to arise when using the PDMA-32:-

1. Memory PARITY CHECK ERROR. This may occur on input mode transfers only if you change the data during the period when the XFER. ACK. is low (active). Your data should be stable prior to issuing a XFER. REQ. and remain stable throughout the period when the transfer is taking place signified by XFER. ACK. being in the low state. Changing the data in this interval might alter it in the middle of a memory write giving the memory insufficient data set up time and hence causing the parity error. Extremely noisy input data can also produce the same problem.

2. XFER. ACK. remains low after the last D.M.A. transfer. This is caused by issuing XFER. REQ's above and beyond the number required by the transfer count programmed into the 8237 D.M.A. controller. Once the specified transfer count is reached, the 8237 D.M.A. controller automatically masks out the D.M.A. channel. Issuing a XFER. REQ. to the PDMA-32 is acknowledged by taking the XFER. ACK. low, but since the channel is inactive, the PDMA-32 is unable to obtain service from the controller and correctly keeps XFER. ACK. low indicating that the transfer has not completed. This condition does not matter, but if you wish to clear it, simply disable D.M.A. from the PDMA-32 by taking bit 7 (DMA ENABLE) of the D.M.A. control register low to disable D.M.A.. This will also set the XFER. ACK. back high i.e. clear the condition.

3. Obtaining the maximum speed. Due to D.M.A. latency, when using constant frequency XFER. REQ's e.g. from the timer, the throughput is limited by the worst case transfer time (close to 5 microseconds). Under synchronous conditions, you can obtain a major improvement (50 - 80%) in throughput by adding a FIFO (first-in first-out) buffer memory to the inputs/outputs of the PDMA-32, you are then limited by the average throughput time rather than the worst case time.

4. First transfer problems. If you write your own setup code for the D.M.A. controllers, it is common to suffer from either missed or doubled first transfers in block transfers. This is indicative of something being wrong in your setup sequence. Examine your steps carefully, once you understand precisely what is happening, you can correct this problem.

5. Extra transfers. An additional transfer at the end of a block is usually caused by programming the 8237 D.M.A. controller with the transfer count rather than the (transfer count - 1).

Section 7

SERVICE & REPAIR

7.1 Service and Repair

No periodic calibration of the PDMA-32 is required. If a digital output or input has been damaged and appears to be faulty you have two options in making a repair. All of the critical integrated circuits that communicate with the external world are in sockets so that they can easily be replaced. The procedure for doing this is detailed in the next section. Alternatively, you can return the board to MetraByte for repair. In this case, first phone our customer service department at (508)-880-3000 and ask for a R.M.A. (return material authorization) number. Then mark the unit with the R.M.A. number and ship it to us any convenient way. It also helps our service department if you can include a brief description of the problem.

7.2 Performing your own Repairs

This series of tests and fixes involves a voltmeter, a scope, and possibly replacement of some standard 74LS series logic available from many manufacturers. All the integrated circuits mentioned are inexpensive and inserted in sockets for easy replacement. To remove a chip, insert a small screwdriver between the chip and socket top and lever out.

The first step is to diagnose the fault. The most likely fault will be damage to an output or input due to static electricity, shorts to power supplies or other over voltages. Check each output port using BASIC or DEBUG:-

```
OUT BASE + 2, 3        'set ports A & B to output mode, byte I/O
OUT BASE, 0            'check all lines port A low
OUT BASE + 1, 0        'check all lines port B low
```

Check all the outputs A0-A7 and B0-B7 with a digital voltmeter. Every output should be between 0 and +0.4v . Next set all the outputs high:-

```
OUT BASE, 255          'set port A outputs high
OUT BASE + 1, 255      'set port B outputs high
```

Use the voltmeter to check that every output is between +2.4v and +5v. Note which outputs are faulty if any. If there are no faults next check the inputs:-

```
PRINT INP(BASE)        'read port A - should return 255
PRINT INP(BASE + 1)    'read port B - should return 255
```

Next output zero again:-

```
OUT BASE, 0
OUT BASE + 1, 0
```

and read back the result:-

```
PRINT INP(BASE)        'read port A - should return 0
PRINT INP(BASE + 1)    'read port B - should return 0
```

If there is any fault in the returned data, note which input line is faulty.

If any of the outputs A0 - A7 are faulty, replace U6 (74LS374) or if outputs are O.K. but any input is faulty replace U5 (74LS244). If any of the outputs B0 - B7 is faulty, replace U11 (74LS374) or if outputs are O.K. but any input is faulty replace U17 & U18 (both 74LS244).

The other inputs and outputs can be checked as follows:-

1. A. DIR. and B. DIR. - write 0 and 3 alternately to the D.M.A. control register at BASE + 2. If either output fails to change state, replace U21, 74LS04. The fault may also be due to a damaged D.M.A. control register chip - see #2 below.

2. AUX 1 and AUX 2 - write 0 and 48 alternately to the D.M.A. control register at BASE + 2. If either output fails to change state, replace D.M.A. control register chip U3, 74LS273. If the AUX 1/2 outputs change state correctly but cannot be read back correctly from the D.M.A. control register, replace U8 (74LS244) as well.

3. AUX 3 - write 0 and 8 alternately to the interrupt control register at BASE + 3. If the output fails to change state, replace U2, 74LS273. IF the AUX1 output cannot be read back correctly from the interrupt control register, replace U7 (74LS244) as well.

4. Run TIMER.BAS and if you cannot see a 1KHz repetition rate pulse on the TIMER OUT, replace U21, 74LS04N. Any other problem with the timer e.g. TIMER GATE inoperative, no pulse with U21 replaced or wrong output frequency, replace U1, 8254 or 82C54.

5. Run DMA.BAS using an auto-initialize input mode and selecting the internal timer as source. The XFER. ACK. should go low for several microseconds at the timer rate, TIMER OUT can be used to synchronize a scope. If XFER. ACK. is stuck low or does not appear to respond, replace U20, 74LS74.

This series of tests will detect the majority of faults caused by damage to the inputs or outputs. All other faults will require return of the board to MetraByte for service.

Appendix A

STORAGE OF INTEGER VARIABLES

The PDMA32.BIN driver makes use of integer variables for transferring data. This appendix explains how to use signed integers (-32768 to 32767) to transfer unsigned data in the range 0 - 65,535.

Data is stored in BASIC's integer variables (% type) in 2's complement form. Each integer variable uses 16 bits or 2 bytes of memory. 16 bits of data is equivalent to values from 0 to 65,535 decimal, but the 2's complement convention interprets the most significant bit as a sign bit so the actual range becomes -32,768 to +32,767 (a span of 65,535). Numbers are represented as follows:-

	High byte								Low byte							
+32,767	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
+10,000	0	0	1	0	0	1	1	1	0	0	0	1	0	0	0	0
+1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
-1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
-10,000	1	1	0	1	1	0	0	0	1	1	1	1	0	0	0	0
-32,768	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

^
Sign bit
1 if negative, 0 if positive

Integer variables are the most compact form of storage for the 16 bit data used for word transfers, byte counts, and timer registers on the PDMA-32. All data exchange via the CALL is through integer type variables. This poses a programming problem when handling unsigned numbers in the range 32,768 to 65,535.

If you wish to input or output an unsigned integer greater than 32,767 then it is necessary to work out what its 2's complement signed equivalent is. As an example, assume we want to load a 16 bit counter with 50,000 decimal. An easy way of turning this to binary is to enter BASIC and execute PRINT HEX$(50000). This returns C350H or binary:-

50,000 (Hex C350) = Binary 1100 0011 0101 0000

Since the most significant bit is 1 this would be stored as a negative integer and in fact the correct integer variable value would be 50,000 - 65,536 = -15,536. The programming steps for switching between integer and real variables for representation of unsigned numbers between 0 and 65,535 is therefore:-

From real variable N (0 <= N <= 65,535) to integer variable N%:-

```
xxx10 IF N<=32767 THEN N% = N ELSE N% = N - 65536
```

From integer variable N% to real variable N:-

```
xxx20 IF N% >= 0 THEN N=N% ELSE N = N% + 65536
```

Appendix B

8254 TIMER DESCRIPTION

B.1 The 8254 Programmable Interval Timer

The Intel 8254 programmable interval timer is used in the PDMA-32. This is a flexible but complex device consisting of three independent 16 bit pre-settable down counters. Each counter can be programmed to divide by any integer in the range 2 - 65,535. As configured in the PDMA-32, the input of counter 0 is connected to an 0.01% precision 10MHz xtal oscillator. The output of counter 0 is connected to both the inputs of counters 1 and 2. Counter 2 output is buffered and inverted and available to the user as the TIMER OUT (pin 22). The other counter, counter 3 has no external connections but it can be loaded and read if required. The primary purpose of the 8254 on the PDMA-32 is as a periodic source for interrupts and D.M.A. transfers, and no attempt has been made to utilize its other modes of operation. The following brief description provides information for programming, the Intel 8254 data sheet provides more comprehensive details.

Each counter has a clock input, a gate input that controls counting and triggering and an output. On the PDMA-32 all the gate inputs are connected together and are available externally on pin 23 (TIMER GATE IN). A block diagram of the 8254 configuration in PDMA-32 is detailed in Fig. 4.1. There are 5 possible operating configurations (called modes) for each counter:-

Mode number	Description
0	PULSE ON TERMINAL COUNT. The output is initially low after setting this configuration. After the count is loaded, the output remains low until the counter decrements thru zero, when it goes high and remains high until the counter is reloaded. The counter will continue to decrement after passing thru zero and counting can be inhibited by a low gate input. This mode produces a single positive going output transition such as may be required in a time delay initiated by the program.
1	PROGRAMMABLE ONE SHOT. The output goes low after a rising edge of the gate input and goes high when the counter passes thru zero. The period that the output is low is set by the loaded count. If the gate input goes high again before the one shot has timed out, a new timing cycle is initiated i.e. the one shot is re-triggerable and if a new count is loaded, it will not become effective until any cycle in

progress has terminated. This provides a hardware triggered delay or one-shot.

2 RATE GENERATOR (or divide by N counter). The output goes low for one input clock period every N counts, where N is the count loaded. The gate input when low, forces the output high, and on going high, reloads the counter. Thus the gate input can be used to synchronize the counter. This mode is the one that is normally selected for operation of the counters on the PDMA-32. The PDMA32.BIN driver initialization mode 0 will automatically set the counters in the rate generator configuration.

3 SQUARE WAVE GENERATOR. This is similar to configuration 2 except that the output is high for half of the count and low for the other half. If the divider integer N is even, a symmetrical square wave output is obtained. If N is odd, the output is high for (N+1)/2 counts and low for (N-1)/2 counts i.e. has a 1 count asymmetry. This configuration can be used in the same way as configuration 2 for periodic triggering or for frequency synthesis and is an alternative operating mode on the PDMA-32.

4 SOFTWARE TRIGGERED STROBE. After the mode is set the output is high. When a count of N is loaded the counter begins counting, and the output will go low for one input clock period as it passes thru zero. The cycle is repeated on loading another count. The gate input may be used to inhibit counting.

5 HARDWARE TRIGGERED STROBE. This is essentially the same as configuration 1, except that the output will go low for one clock period at the end of the cycle and return high again. The start of the cycle is triggered by the rising edge of the gate input, and as in configuration 1, is re-triggerable.

As you can see from the possible counter configurations, you may find other configurations useful apart from the default rate generator mode set by the PDMA32.BIN driver. If you wish to program these other modes, you will have to read and write directly to the counter I/O ports.

The 8254 programmable interval counter occupies 4 I/O address locations:-

Address			Register type	Description
BASE ADDRESS	+	4	Read/write	Counter 0
" "	+	5	Read/write	Counter 1
" "	+	6	Read/write	Counter 2
" "	+	7	Write only	Control

Before loading or reading any of the individual counters, the control register must be loaded with data setting the counter operating configuration as above, the type of read or write operation that will be performed (see following) and the modulus, binary (0 - 65,535) or BCD (Binary coded decimal 0 - 9,999). The format of the control byte is:-

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	RL1	RL0	M2	M1	M0	BCD

SC1-0 - Control which counter is selected.

SC1	SC0	Counter
0	0	0
0	1	1
1	0	2
1	1	Read back command (see next section)

RL1-0 - Control the type of read/load operation.

RL1	RL0	Operation
0	0	Counter latch (see following)
0	1	Read/load most significant byte
1	0	Read/load least significant byte
1	1	Read/load least significant byte, followed by most significant byte.

M2-0 - Control counter configuration as above.

M2	M1	M0	Configuration
0	0	0	0 - Pulse on terminal count
0	0	1	1 - Programmable one shot
X	1	0	2 - Rate generator
X	1	1	3 - Square wave generator
1	0	0	4 - Software triggered strobe
1	0	1	5 - Hardware triggered strobe

BCD - Controls binary/decimal counting.

BCD	Counter type
0	Binary 16 bits
1	Decimal 4 decades

The counters may be programmed to count in binary (modulus 2) or binary coded decimal (modulus 10) modes by the BCD bit. The binary mode with a full count of 65,535 has the obvious advantage of providing a larger count range than the BCD mode which has a 9,999 full scale and is also compatible with the internal storage of binary integers used by the PC.

B.2 Reading and Loading the Counters

If you attempt to read the counters on the fly with a high input frequency, you will most likely obtain erroneous data. This is partly caused by the rippling of the counter during the read and

also by the fact that the low and high bytes are read sequentially rather than simultaneously, making it highly probable that carries will be propagated from the low to high byte during the read cycle. To circumvent these problems, you can perform a counter latch operation in advance of the read cycle. To do this you load the RL 1/0 bits of the control byte with "0 0" which instantaneously latches the count of the selected counter in a 16 bit hold register. An alternative method a latching counter(s) is by use of the read back command - see below. A subsequent read operation on the selected counter returns the held value. Latching is the only way of correctly reading a counter "on the fly" without disturbing the counting process. If you do not specify a latching operation, then the counter itself will be read. You can only rely on direct data if the counting operation is suspended while reading e.g. by taking the TIMER GATE low.

For each counter you are required to specify in advance the type of read or load operation that you intend to perform. You have a choice of loading/reading the high byte of the count or the low byte of the count, or the low byte followed by the high byte. This last mode is of the most general use and is selected for each counter by setting the RL 1/0 bits to "1 1". Subsequent read/load operations must be performed in pairs in this sequence, otherwise the internal sequencing flip-flop of the 8253 will get out of step.

If the SC0 and SC1 bits are both set to 1, you can perform two types of operations. When CNT=0 (see below) counters selected by C0 thru C2 are latched **simultaneously**. When STA = 0, the counter status byte will be read on accessing the counter I/O location. The status byte provides information on the current output state of the counter, and its operating mode.

Read back command byte format:-

D7	D6	D5	D4	D3	D2	D1	D0
SC1	SC0	CNT	STA	C2	C1	C0	0
1	1						

Read back command

CNT - When 0, latches counters selected by C0 - C2.

STA - When 0, returns status byte of counters selected by C0 - C2.

C0 - C2 - When high, select a particular counter for a read back operation:-

C0 = 1 selects Counter 0
C1 = 1 selects Counter 1
C2 = 1 selects Counter 2

The status byte returned if STA = 0, consists of:-

D7	D6	D5	D4	D3	D2	D1	D0
OUT	NC	RL1	RL0	M2	M1	M0	BCD

OUT - Current state of counter output

NC - Null count. This indicates when the last count loaded into the counter register has actually been loaded into the counter itself. The exact time of load depends on the configuration selected. Until the count is loaded into the counter itself, it cannot be read from the counter.

If both the STA and CNT bits are set low and RL0 and RL1 have both been previously set high in the counter control byte (selecting 2 byte reads) then reading a selected counter location will yield:-

1st. read : Status byte
2nd. read : Low byte of latched data
3rd. read : High byte of latched data
4th. read : Low byte of counter direct
5th. read : High byte of counter direct

Some programming examples in BASIC:-

```
xxx10 OUT BASE + 7, &H36        'set Counter 0 to square wave mode
xxx20 OUT BASE + 7, &H76        'set Counter 1 to square wave mode
xxx30 X = 1000                  'number to load both counters
xxx40 XH% = INT(X/256)          'high byte
xxx50 XL% = X - 256 * XH%       'low byte
xxx60 OUT BASE + 4, XL%
xxx70 OUT BASE + 4, XH%         'load counter 0
xxx80 OUT BASE + 5, XL%
xxx90 OUT BASE + 5, XH%         'load counter 1
```

TIMER OUT will now be outputting a 10Hz. square wave.

The following lines would determine the status of counter 1:-

```
yyy10 OUT BASE + 7, &HE4            'control byte to read status
yyy20 PRINT HEX$(INP(BASE + 5))     'read status
```

Appendix C

SPECIFICATIONS

+5v power	:	850mA typical / 1 A max.
-5v, +12v, -12v power	:	Not used
I/O ports	:	Ports A & B - each of 8 bits Direction software selectable. Auxiliary - 3 outputs
Logic output levels	:	Ports A & B - TTL/DTL:- 0.4v max low at Isink = 24mA 2.4v min high at Isource = -2.6mA All other outputs TTL/DTL:- 0.4v max low at Isink = 8mA 2.4v min high at Isource = -400uA
Logic input levels	:	All inputs TTL/DTL compatible 0.8v max low level at -400uA 2.0v min high level at 20uA
Timer	:	Internal 8254 timer & 10MHz xtal clock. Pulse rates from 2.5Mhz to 0.002 Hz.
I/O address	:	Can be set on any 16 bit boundary from Hex 0 to Hex 3F0 (Hex 100 - 3F0 useable in PC/AT)
PC bus internal loading	:	One 74LS TTL load on all inputs
Interrupts	:	Software selectable on levels 3 - 7, 9 - 11 & 14 - 15. Active when interrupt enabled, tri-state otherwise.
Interrupt source	:	1 of 3 software selectable:- External input +/- slope PDMA-32 timer D.M.A. terminal pulse

D.M.A. level	:	Software selectable:- Byte transfer levels 0 - 3 Word transfer levels 5 - 7 (only active in DMA transfer, tri-state otherwise)
D.M.A. transfer source	:	1 of 2 software selectable:- External request PDMA-32 timer
D.M.A. transfer rate	:	350,000 transfers/sec asynchronous. 200,000 transfers/sec clocked synchronous.
Maximum number of PDMA-32's in one computer	:	Limited only by availability of expansion slots and/or D.M.A. or interrupt levels for concurrent use.
Connector	:	On rear plate, 37 pin D type male
User adjustments /calibration	:	None required
Card size	:	13.2" long x 3.9" high full size PC/AT connector
Weight	:	6 oz. (170 gm.)
Operating temperature range	:	0 to 60 deg. C.
Storage temperature range	:	-40 to 100 deg. C.
Humidity	:	0 - 90% non-condensing

Appendix D

COMPARISON OF PDMA-32 AND PDMA-16

D.1 General Comments

The PDMA-32 for the PC/AT is essentially functionally identical to MetraByte's PDMA-16 for the PC, PC/XT & PC/AT. It's main advantage is that it can perform true word transfers (16 bit bus) both in D.M.A. and programmed I/O and is faster than a PDMA-16 when used in these modes. It also offers a wider choice of operating D.M.A. and interrupt levels.

In most cases upgrading from the PDMA-16 to the PDMA-32 will simply involve a software change requiring substitution of the PDMA32.BIN driver for the PDMA.BIN driver. The minor differences that do exist are detailed in this appendix.

D.2 Pin Connections and Functions

The rear pin out and signal functions of the PDMA-32 and PDMA-16 are identical.

The operating BASE ADDRESS of the PDMA-32 is selected by a 6 position dip switch similar to the 7 position switch of the PDMA-16, just ignore the missing 7th. slider. The I/O map now uses 16 locations instead of 8, extra registers are used for DMA & interrupt level select and interrupt status. The first 8 I/O addresses of both boards have the same registers and functions and so remain compatible i.e. the PDMA-16's registers are a subset of the PDMA-32's.

D.3 DMA Functions

The PDMA-16 could only operate on 2 D.M.A. levels, 1 or 3 and could only perform byte (8 bit wide) transfers. It "simulated" word transfers by internally performing 2 consecutive byte transfers. The maximum number of transfers (controlled by the PC's 8237 DMA controller) was 65,536 so that the limit was 65,536 bytes or 32,768 words of data maximum per transfer. The PDMA-32 performs single transfers in both byte and word mode and so will be faster in word mode than a PDMA-16. There is no equivalent double byte transfer mode for words available on the PDMA-32, word transfers are made more efficiently on the PC/AT's word DMA transfer levels 5 - 7.

PC/AT type machines include 2 cascaded 8237 DMA controllers. Levels 0 - 3 provide byte transfers on controller #1, whereas levels 5 - 7 provide word transfers on controller #2. Note that

the system logic is hard wired so you cannot alter this configuration, specifically byte controller #1 sits across address lines A0-A15 and word controller #2 sits across address lines A1-A16. This means that both controllers can make up to 65,536 transfers, either 65,536 byte transfers or 65,536 word transfers, so the PC/AT can transfer twice as many words using levels 5 - 7 in a single transfer as a standard PC can on levels 1 or 3 (actually 128 Kbytes). When you are programming, watch out for a few quirks, registers on controller #2 are addressed at every other address (since there is no connection to A0) and the least significant bit of the page register is irrelevant as it is supplanted by the most significant address bit from the DMA controller.

The PDMA-32 gives you the option of using DMA level 2 which was not available on the PDMA-16. If you use this level, be aware that it is usually used by the floppy diskette controller, to share this level you must write a bit in one of the diskette controllers registers to disable it (OUT &H3F2, &H04) before enabling the PDMA-32 on level 2. DOS will re-enable the diskette controller on any disk activity.

The word levels 5 - 7 are generally free, but one of them could be used by the hard disk. Check with the specifications of your particular computer. Most hard disk controllers no longer use DMA to transfer data as it is actually slower than block string moves on 286/386 machines.

To accommodate the added DMA levels, the PDMA-32 includes a new DMA level select register at BASE + 8. This is a read/write register, but is not initialized to any specific state on power up. Be sure to initialize it with your selected level before enabling DMA (mode 0 of the driver does this). You can always read this register to find out what level the board is operating on which you will need for controller setup purposes. The old DMA control register at I/O address BASE + 2 is cleared on power up, disabling DMA and putting ports in the input mode. The PDMA-16's DMA level select bit (bit D6) will always read back as zero on the PDMA-32 regardless of what is written to it.

D.4 I/O ports

The I/O ports PA and PB may be operated as separate byte oriented ports or a single word oriented port for programmed (non-DMA) I/O purposes. A number of possible combinations are possible depending on the port mode set, bit D2 of the DMA control register, and the DMA level selected as follows:-

Bit D2	DMA Level	Port I/O	A & B direction
0 - byte	0 - 3 (byte)	Byte for I/O Byte for D.M.A.	A & B independent
1 - word	0 - 3 (byte)	Word for I/O Byte (PA) for D.M.A.	A controls both ports B irrelevant
0 - byte	5 - 7 (word)	Byte for I/O Word for D.M.A.	A controls both ports B irrelevant
1 - word	5 - 7 (word)	Word for I/O Word for D.M.A.	A controls both ports B irrelevant

For normal programmed (non-D.M.A.) byte I/O use IN AL,DX or OUT DX,AL (assembly) or INP(---) or OUT ---, --- in BASIC to I/O addresses BASE or BASE + 1 and set the ports in byte I/O mode.

Setting bit D2 of the DMA control register to 1 will weld the 2 ports into a single 16 bit wide word oriented port, with its direction controlled by the A direction bit, bit D0, the B direction bit, bit D1, is ignored in this case. You can now read or write the whole port with a single 16 bit I/O instruction such as IN AX,DX or OUT DX,AX (assembly) to address BASE. In BASIC there are no 16 bit I/O instructions so forget it or use modes 4 and 5 of the driver instead.

What happens in D.M.A.? Well if the DMA is byte oriented (levels 0 - 3) there is no problem, DMA takes place through the PA port and the PB port is unaffected, it does not matter whether the ports are programmed for normal I/O in word or byte mode. If the DMA is word oriented and the ports have been set up as bytes for normal I/O, the D.M.A. data direction will control the direction of both ports. You still have the choice through bit D2 of doing byte or word reads of the PA and PB ports for normal I/O (non-D.M.A.) but remember that the selected DMA data direction overrides all else. This seems to be the most logical compromise of providing either word or byte I/O access to the ports together with D.M.A.. The hardware explanation behind all this is that bit D2 controls whether the PDMA-32 appears to the PC/AT bus for normal I/O as a 16 bit or 8 bit peripheral as far as access to the PA and PB ports are concerned. On 16 bit accesses, bus signals -SBHE and -I/O CS16 have to be asserted, on 8 bit accesses they are not, also the data is shifted from the D8-D15 data bits to the D0-D7 data bits.

D.5 Interrupts

To accommodate the expanded interrupt capabilities of the PC/AT bus, an interrupt level select register has been added at I/O address BASE + 9. The PDMA-16 could operate on levels 2-7, the PDMA-32 can operate on levels 3-7, 9-12 and 14-15. Note that level 2 is re-vectored to level 9 on the PC/AT bus. Level 6 is used by the floppy disk controller and level 14 is used by the hard disk controller on the PC/AT, so it is best to avoid using them although the board can be set up on them. The interrupt level select register is a read/write register (handy for setup) but is not initialized to any specific value on power up. Be sure to initialize it before enabling interrupts. The interrupt control register at BASE + 3 is cleared on power up to disable interrupts. All the bit functions are the same as the PDMA-16 except that the PDMA-16's old interrupt level select bits D4 through D6 will read back as 0 on the PDMA-32 regardless of what is written to them.

The PDMA-16 occasionally had problems with interrupts because the interrupt request to the bus was generated using a monostable. The 8259 interrupt controller is programmed to respond to positive edges on the interrupt request line. If you read the small print in the Intel data sheet, you will discover that even though it is edge triggered, the interrupt request must be maintained in its high state until the interrupt service routine has started (interrupt granted). Occasionally on the PDMA-16, the interrupt request pulse would disappear before the processor got round to servicing the interrupt (due to other pending interrupts) and the interrupt would be lost. On the PDMA-32, this problem has been corrected by using a latch for the interrupt request instead of a monostable. Your interrupt service routine must explicitly clear this latch by reading BASE + 10, the interrupt status register, to re-enable interrupts from the PDMA-32. You must also reckon with the added complexity of issuing 2 EOI's to the cascaded 8259 controllers in your service routine. An example is provided in the PDMA32.ASM driver. The additional levels also make the interrupt setup a bit more complicated - see mode 7.

D.6 Driver Software

The PDMA32.BIN driver has purposely been kept as identical as possible to the PDMA-16 driver. The modes 0 - 8 and the call statement structure and parameters are the same. These are the few small differences:-

1. Mode 0 - initialization. An additional parameter D%(3) specifies whether ports PA and PB are to be treated as separate byte I/O ports or a single word I/O port. In effect this tells the PC/AT whether to treat the PDMA-32 as a 16 bit or 8 bit peripheral as far as PA/PB port access is concerned, hence it is included in the initialization mode. All other registers are treated as byte I/O. Also mode 0 checks the "signature" of the PDMA-32 and will return error 4 if not found.

2. Mode 1 - D.M.A. setup. This has been expanded to include additional levels. Parameter D%(1) which used to specify byte or word mode transfers on the PDMA-16 is irrelevant and is ignored on the PDMA-32. This data is now derived from the selected DMA level, 0-3 are assumed byte, 5-7 are assumed word.

3. Mode 2 - Read status. Works the same as the PDMA-16 except that we found that it was a good idea to mask the DMA controller before reading the current word count, and unmask it afterwards. This avoids getting the wrong answer on occasion. It also has been expanded to handle both DMA controllers.

4. Mode 3 - Set timer. This is unchanged.

5. Mode 4 - Digital Output. Expanded to include 16 bit word I/O and D%(1) the port selector, is only relevant if ports are in byte mode (set by mode 0).

6. Mode 5 - Digital Input. Expanded to include 16 bit word I/O and D%(1) the port selector, is only relevant if ports are in byte mode (set by mode 0).

7. Mode 6 - Auxiliary I/O. This is unchanged.

8. Mode 7 - Install and Enable Interrupt. Expanded to handle additional interrupts and setup. Also see example of expanded interrupt handler.

9. Mode 8 - Disable interrupt. Expanded to handle additional interrupts.

D.7 Other Comments

If you are in the position of writing one piece of software to work with both PDMA-16's and PDMA-32's i.e. be equipment independent, it is not too difficult to accomplish. You can establish whether you are dealing with a PDMA-16 or PDMA-32 by read/writing the extra registers on the PDMA-32 at BASE + 8 & 9 and writing to the zero bits in the DMA and interrupt control registers at BASE + 2 & 3. This will give you a "signature" of which board is installed, you can then select the appropriate driver or setup routine and since all the rear connections are the same, the cabling stays identical.

Index

8237 D.M.A. controller 3-6
8254 timer description B-1

A

A/D conversion 6-4
Applications - typical 6-1
Assembly language programming 5-23

B

Base I/O address 2-1
BASIC - memory map with driver installed 5-2

C

CALL routine: Format 5-3
Compiling a BASIC program 5-22
Conflict with other peripherals 2-1
Connection examples 6-3
Connections 4-8
Counter-timer B-1

D

D.M.A. 3-1
D.M.A. - determining status of a transfer 3-15
D.M.A. levels 3-3
D.M.A. transfer setup 3-12
D.M.A. transfer timing 3-14
Disk back up 2-1

E

Example programs 5-22

G

General description of PDMA-32 1-1

H

Hardware specification C-1

I

Initialization 5-7
INSTALL.EXE switch setting aid 2-1
Installation - hardware 2-1
Integer variables - 2's complement storage A-1
Interrupts - use with D.M.A. 3-15

M

Memory map with PDMA32.BIN driver 5-2
Mode 0 5-7
Mode 1 5-8
Mode 2 5-11
Mode 3 5-12
Mode 4 5-13
Mode 5 5-15
Mode 6 5-17
Mode 7 5-18
Mode 8 5-21
Mode number 5-6
Multiple PDMA-32 boards 5-23

P

Page registers 3-11
PDMA-16 compared to PDMA-32 D-1
PDMA-16 to PDMA-32 upgrade C-2
PDMA32.ASM - changing the driver 5-23
PDMA32.BIN 5-1
PDMA32.BIN error codes 5-5
Programming 5-1

R

Registers 4-1
Repairs 7-1

S

Software license conditions 1-3
Specifications C-1
Standard I/O addresses 2-1
Storage out of computer 2-2
Switch setting 2-1

T

Timer description B-1
Typical hardware connection 6-1

U

Unresolved external error 5-23

W

Waveform generation 6-3